Санкт-Петербургский Политехнический Университет Петра Великого

Кафедра компьютерных систем и программных технологий

Отчёт по лабораторной работе №4

Курс: «Операционные системы»

Тема: «Системное программирование UNIX»

Выполнил студент:

Волкова М.Д.
Группа: 43501/3

Проверил:

Малышев И.А.

Санкт-Петербург
2017 г.

Лабораторная работа №4

1.1 Цель работы

Изучение средств межпроцессорного взаимодействия (IPC) в ОС Linux, таких как: надежные и ненадежные сигналы, именованные и неименованные каналы, очереди сообщений, семафоры и разделяемая память, сокеты.

1.2 Программа работы

Глава 1. Ненадежные сигналы

1. Составьте программу, позволяющую изменить диспозицию сигналов, а именно, установить: обработчик пользовательских сигналов SIGUSR1 и SIGUSR2, реакцию по умолчанию на сигнал SIGINT, игнорирование сигнала SIGCHLD. Породить процесс копию и уйти в ожидание. Обработчик сигналов должен содержать восстановление диспозиции и оповещение на экране о (удачно или неудачно) полученном сигнале и идентификаторе родительского процесса. Процесс-потомок, получив идентификатор родительского процесса, посылает процессу-родителю сигнал SIGUSR1 и извещает об удачной или неудачной отправке указанного сигнала. Остальные сигналы генерируютя из командной строки.

2. Повторить эксперимент для других сигналов, порождаемых в разных файлах, а так же для потоков одного и разных процессов.

Глава 2. Надежные сигналы

1. Создать программу, позволяющую продемонстрировать возможность отложенной обработки (временного блокирования) сигнала.

2. Изменить обработчик так, чтобы отправка сигнала SIGINT производилась из обработчика функцией kill.

3. Сравните обработчики надежных и ненадежных сигналов.

Глава 3. Сигналы POSIX реального времени

1. Провести эксперимент, позволяющий определить возможность организации очереди для различных типов сигналов, обычных и реального времени.

2. Экспериментально подтвердить, что обработка равноприоритетных сигналов реального времени происходит в порядке очереди.

3. Опытным путем определить наличие приоритетов сигналов реального времени.

Глава 4. Неименованные каналы

1. Организовать программу так, чтобы процесс-родитель создавал неименованный канал, создавал потомка, закрывал канал на запись и записывал в произвольный текстовый файл считываемую информацию. В процессе-потомке будет происходить считывание данных из файла и запись в канал.

Глава 5. Именованные каналы

1. Создать клиент-серверное приложение, демонстрирующее дуплексную передачу информации двумя однонаправленными именованными каналами между клиентом и сервером.

Глава 6. Очереди сообщений

1. Создать клиент-серверное приложение, демонстрирующее передачу информации между процессами посредством очереди сообщений.

Глава 7. Семафоры и разделяемая память

1. Есть один процесс, выполняющий запись в разделяемую память и один процесс, выполняющий чтение из нее. Под чтением понимается извлечение данных из памяти. Программа должна обеспечить невозможность повторного чтения одних и тех же данных и невозможность перезаписи данных, т.е. новой записи, до тех пор, пока читатель не прочитает предыдущую. В таком варианте задания для синхронизации процессов достаточно двух семафоров.

2. Добавление условия корректной работы нескольких читателей и нескольких писателей одновременно.

3. Добавление не единичного буфера, а буфера некоторого размера. Тип буфера не имеет значения.

Глава 8. Сокеты

1. Реализовать TCP сервер, который прослушивает заданный порт. При приходе нового соединения создается новый поток для его обработки. Работа с клиентом организована как бесконечный цикл, в котором выполняется прием сообщений от клиента, вывод его на экран и пересылка обратно клиенту. Клиентская программа после установления с сервером так же в бесконечном цикле выполняет чтение ввода пользователя, пересылку его серверу и получение сообщения. Если была введена пустая строка, клиент завершает работу.

2. Модифицировать предыдущую программу для работы с большим количеством клиентов. Провести эксперимент, определяющий при каком максимальнов количестве клиентов TCP сервер завершает работу.

3. Выполнить аналогичные действия на основе протокола UDP, сравнить с очередями сообщений.

4. Провести эксперимент, определяющий при каком максимальнов количестве клиентов UDP сервер завершает работу.

5. Запустить клиент-серверные приложения на лабораторных компьютерах.

1.3 Ход работы

1.3.1 Глава 1. Ненадежные сигналы

1. Составьте программу, позволяющую изменить диспозицию сигналов, а именно, установить: обработчик пользовательских сигналов SIGUSR1 и SIGUSR2, реакцию по умолчанию на сигнал SIGINT, игнорирование сигнала SIGCHLD. Породить процесс копию и уйти в ожидание.

Разработаем программу, выполняющую эти операции:

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

// Обработчик сигнала
static void handler(int sig);

int main() {
  printf("Parent pid %d, ppid %d.\n", getpid(), getppid());

  // Задаем сигналам SIGUSR1, SIGUSR2 одинаковый обработчик сигнала
  signal(SIGUSR1, handler);
  signal(SIGUSR2, handler);
  // Сигнал прерывания по умолчанию
  signal(SIGINT, SIG_DFL);
  // Сигнал прерывания игнорируется
  signal(SIGCHLD, SIG_IGN);

  if(!fork()) {
```

```
21      printf("Child pid %d, ppid %d.\n", getpid(), getppid());
22      // Отправка прерывания родительскому процессу
23      if(kill(getppid(), SIGUSR1)) {
24        perror("It's impossible to send SIGUSR1.\n");
25        exit(0x1);
26      }
27
28      printf("Signal has been successfully sent.\n");
29      return 0x0;
30    }
31
32    // Ожидание сигнала
33    while(1)
34      pause();
35
36    printf("Exit message.\n");
37
38    return 0x0;
39  }
40
41  static void handler(int sig) {
42    printf("Signal handle pid %d, ppid %d.\n", getpid(), getppid());
43
44    // Возвращаем обработчик по умолчанию
45    signal(sig, SIG_DFL);
46  }
```

Результат выполнения программы:

```
1 stakenschneider@stakenschneider:~/temp$ gcc p1.1.c -o p1.1
2 stakenschneider@stakenschneider:~/temp$ sudo ./p1.1
3 Parent pid 2100, ppid 2099.
4 Child pid 2101, ppid 2100.
5 Signal has been successfully sent.
6 Signal handle pid 2100, ppid 2099.
7 ( ... )
```

Процесс-потомок успешно послал родительскому процессу сигнал *SIGUSR1*, что видно из результатов программы. После этого сигнал был успешно обработан.

Теперь из другого терминала отправим сигнал *SIGUSR2*:

```
1 # Terminal 1
2 stakenschneider@stakenschneider:~/temp$ sudo ./p1.1
3 Parent pid 2100, ppid 2099.
4 Child pid 2101, ppid 2100.
5 Signal has been successfully sent.
6 Signal handle pid 2100, ppid 2099.
7 ( ... )
8
9 # Terminal 2
10 stakenschneider@stakenschneider:~$ sudo kill -SIGUSR2 2326
11
12 # Terminal 1
13 stakenschneider@stakenschneider:~/temp$ sudo ./p1.1
14 Parent pid 2326, ppid 2325.
15 Child pid 2327, ppid 2326.
16 Signal has been successfully sent.
17 Signal handle pid 2326, ppid 2325.
18 Signal handle pid 2326, ppid 2325.
19 ( ... )
```

Кроме сигнала *SIGUSR1*, отправленного процессом-потомком, был принят также и сигнал *SIGUSR2*, отправленный с терминала, о чем свидетельствует два вывода "Signal handle ...".

Убедимся, что сигнал *SIGCHLD* игнорируется:

```
1 # Terminal 1
2 stakenschneider@stakenschneider:~/temp$ sudo ./p1.1
```

```
Parent pid 2458, ppid 2457.
Child pid 2459, ppid 2458.
Signal has been successfully sent.
Signal handle pid 2458, ppid 2457.
( ... )

# Terminal 2
stakenschneider@stakenschneider:~$ sudo kill -SIGCHLD 2458

# Terminal 1
stakenschneider@stakenschneider:~/temp$ sudo ./p1.1
Parent pid 2458, ppid 2457.
Child pid 2459, ppid 2458.
Signal has been successfully sent.
Signal handle pid 2458, ppid 2457.
( ... )
```

После отправки сигнала *SIGCHLD* со второго терминала, на первом терминале ничего не изменилось, что означает игнорирование сигнала.

Убедимся, что сигнал *SIGINT* завершает программу:

```
# Terminal 1
stakenschneider@stakenschneider:~/temp$ sudo ./p1.1
Parent pid 2480, ppid 2479.
Child pid 2481, ppid 2480.
Signal has been successfully sent.
Signal handle pid 2480, ppid 2479.
( ... )

# Terminal 2
stakenschneider@stakenschneider:~$ sudo kill -SIGINT 2480

# Terminal 1
stakenschneider@stakenschneider:~/temp$ sudo ./p1.1
Parent pid 2480, ppid 2479.
Child pid 2481, ppid 2480.
Signal has been successfully sent.
Signal handle pid 2480, ppid 2479.
stakenschneider@stakenschneider:~/temp$
```

После отправки сигнала *SIGINT* со второго терминала, приложение завершилось, что соответствует действию сигнала по умолчанию.

Отправим два сигнала *SIGUSR2*:

```
stakenschneider@stakenschneider:~/temp$ sudo ./p1.1 &
[1] 2661
stakenschneider@stakenschneider:~/temp$ Parent pid 2662, ppid 2661.
Child pid 2663, ppid 2662.
Signal has been successfully sent.
Signal handle pid 2662, ppid 2661.

stakenschneider@stakenschneider:~/temp$ jobs
[1]+  Running                 sudo ./p1.1 &
stakenschneider@stakenschneider:~/temp$ sudo kill -SIGUSR2 2662
Signal handle pid 2662, ppid 2661.
stakenschneider@stakenschneider:~/temp$ sudo kill -SIGUSR2 2662
[1]+  Done                    sudo ./p1.1
```

В результате видно, что первый сигнал был перехвачен обработчиков, а второй - нет. Это связано с тем, что после первого сигнала был восстановлен обработчик по умолчанию.

2. Повторить эксперимент для других сигналов, порождаемых в разных файлах, а так же для потоков одного и разных процессов.

Была создана программа, порождающая поток, в котором был создан обработчик сигнала *SIGUSR2*. Обработчик сигнала содержит код, который должен завершать поток:

```c
#include <stdio.h>
#include <pthread.h>
#include <signal.h>
#include <unistd.h>

#define WHILE_DELAY 1
#define KILL_DELAY 1

// Обработчик сигнала
void signalHandler();
// Обработчик потока
void* threadHandler(void*);

int main() {
  pthread_t thread;

  // Создаем новый поток
  pthread_create(&thread, NULL, &threadHandler, NULL);

  while(1) {
    printf("Main thread message.\n");
    sleep(WHILE_DELAY);
  }
}

void signalHandler() {
  printf("Signal handle.\n");

  // Завершаем текущий поток из обработчика сигнала
  pthread_exit(0);
}

void* threadHandler(void* ptr) {
  // Установка обработчика сигнала
  signal(SIGUSR2, signalHandler);

  sleep(KILL_DELAY);

  // Отправление сигнала прерывания
  kill(getpid(), SIGUSR2);

  while(1) {
    printf("Thread handler message.\n");
    sleep(WHILE_DELAY);
  }

}
```

Результат работы программы:

```
stakenschneider@stakenschneider:~/temp$ gcc -pthread p1.2.1.c -o p1.2.1
stakenschneider@stakenschneider:~/temp$ ./p1.2.1
Main thread message.
Main thread message.
Signal handle.
Thread handler message.
Thread handler message.
Thread handler message.
```

Хоть привязка сигнала к обработчику и происходит в побочном потоке, пришедший сигнал завершил основной поток. Для контроля побочного потока необходимо использовать системный вызов *pthread_kill*.

Исправим программу, заменив функцию *kill* на *pthread_kill*:

```c
#include <stdio.h>
#include <pthread.h>
#include <signal.h>
```

```c
#include <unistd.h>

#define WHILE_DELAY 1
#define KILL_DELAY 1

// Обработчик сигнала
void signalHandler();
// Обработчик потока
void* threadHandler(void*);

int main() {
  pthread_t thread;

  // Создаем новый поток
  pthread_create(&thread, NULL, &threadHandler, NULL);

  sleep(KILL_DELAY);

  // Отправление сигнала прерывания
  pthread_kill(thread, SIGUSR2);

  while(1) {
    printf("Main thread message.\n");
    sleep(WHILE_DELAY);
  }
}

void signalHandler() {
  printf("Signal handle.\n");

  // Завершаем текущий поток из обработчика сигнала
  pthread_exit(0);
}

void* threadHandler(void* ptr) {
  // Установка обработчика сигнала
  signal(SIGUSR2, signalHandler);

  while(1) {
    printf("Thread handler message.\n");
    sleep(WHILE_DELAY);
  }

}
```

Результат работы программы:

```
stakenschneider@stakenschneider:~/temp$ gcc -pthread p1.2.2.c -o p1.2.2
stakenschneider@stakenschneider:~/temp$ ./p1.2.2
Thread handler message.
Main thread message.
Signal handle.
Main thread message.
Main thread message.
Main thread message.
```

Теперь, как и ожидалось, завершается побочный поток, а основной продолжает свою работу.

Напишем программу-родитель и программу-предок, с помощью который определить задержку между отправкой сигнала одним процессом и приемом его другим процессом.

Родительский процесс содержит обработчик сигнала с выводом времени:

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>
```

```
7
8  // Обработчик сигнала
9  void handler() {
10   // Вывод текущего системного времени
11   char buffer[100];
12   struct timeval timeV;
13   gettimeofday(&timeV, NULL);
14   time_t currentTime = timeV.tv_sec;
15   strftime(buffer, 100, "%T", localtime(&currentTime));
16
17   printf("%s.%.3ld Signal handle.\n", buffer, timeV.tv_usec);
18
19   exit(0);
20 }
21
22 int main() {
23   printf("Parent pid %d, ppid %d.\n", getpid(), getppid());
24
25   // Задаем обработчик сигнала
26   signal(SIGUSR1, handler);
27   while(1);
28 }
```

Второй процесс содержит отправку сигнала с фиксацией времени (*PID* передается через аргумент командной строки):

```
1  #include <stdio.h>
2  #include <signal.h>
3  #include <unistd.h>
4  #include <stdlib.h>
5  #include <sys/time.h>
6  #include <time.h>
7
8  int main(int argc, char** argv) {
9    // Вывод текущего системного времени
10   char buffer[100];
11   struct timeval timeV;
12   gettimeofday(&timeV, NULL);
13   time_t currentTime = timeV.tv_sec;
14   strftime(buffer, 100, "%T", localtime(&currentTime));
15
16   printf("%s.%.3ld Signal sent.\n", buffer, timeV.tv_usec);
17
18   // Посылаем сигнал прерывания процессу, чей pid задан извне
19   kill(atoi(argv[1]), SIGUSR1);
20 }
```

Результат работы программы:

```
1  # Terminal 1
2  stakenschneider@stakenschneider:~/temp$ gcc -pthread p1.2.3.c -o p1.2.3
3  stakenschneider@stakenschneider:~/temp$ gcc -pthread p1.2.3.pr.c -o p1.2.3.pr
4  stakenschneider@stakenschneider:~/temp$ ./p1.2.3
5  Parent pid 3594, ppid 1987.
6  ( ... )
7
8  # Terminal 2
9  stakenschneider@stakenschneider:~/temp$ ./p1.2.3.pr 3594
10 13:41:32.563475 Signal sent.
11
12 # Terminal 1
13 stakenschneider@stakenschneider:~/temp$ ./p1.2.3
14 Parent pid 3594, ppid 1987.
15 13:41:32.563895 Signal handle.
```

Фиксация времени отправления и доставки была произведена с точностью до шести знаков после запятой. Задержка между отправлением и приемом сигнала составила 0.42 миллисекунды, что является достаточно

хорошим результатом по производительности. Из этого можно сделать вывод, что обмен данными между двумя процессами посредством сигналов - это быстрая операция, которую можно применять на практике.

1.3.2 Глава 2. Надежные сигналы

1. Провести эксперимент, позволяющий определить возможность организации очереди для различных типов сигналов, обычных и реального времени.

Разработаем программу, которая организует очередь из сигналов, с помощью следующих функций:

```
int sigemptyset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

Функция *sigemptyset* инициализирует набор сигналов, указанный в *set*, и очищает его от всех сигналов. Функция *sigaddset* добавляет сигнал *signum* к *set*.

Системный вызов *sigaction* используется для изменения действий процесса при получении соответствующего сигнала. Параметр *signum* задает номер сигнала и может быть равен любому номеру, кроме *SIGKILL* и *SIGSTOP*. Если параметр *act* не равен нулю, то новое действие, связянное с сигналом *signum*, устанавливается соответственно *act*. Если *oldact* не равен нулю, то предыдущее действие записывается в *oldact*.

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

// Задержка перед следующим сигналом
#define DELAY 10

// Функция, изменяющая набор сигналов
void (*specialSignal(int sig, void (*handler) (int))) (int);
// Обработчик сигнала
void userHandler(int sig);

int main() {
  specialSignal(SIGUSR1, userHandler);

  while(1)
    pause();

  return 0;
}

void (*specialSignal(int sig, void (*handler) (int))) (int) {
  struct sigaction action;
  // Задаем обработчик сигнала
  action.sa_handler = handler;
  // Инициализируем набор сигналов
  sigemptyset(&action.sa_mask);
  // Добавляем сигнал прерывания SIGINT в набор
  sigaddset(&action.sa_mask, SIGINT);
  // Инициализируем флаг нулем
  action.sa_flags = 0;
  // Изменим действие процесса при получении сигнала прерывания
  if(sigaction(sig, &action, 0) < 0)
    return SIG_ERR;

  return action.sa_handler;
}

void userHandler(int sig) {
  // Обрабатываем прерывание

```

```
46    // Если не тот сигнал
47    if(sig != SIGUSR1) {
48      perror("Wrong type of signal.\n");
49      return;
50    }
51
52    printf("SIGUSR catched.\n");
53
54    // Устанавливаем задержку перед обработкой следующего сигнала
55    sleep(DELAY);
56  }
```

Результат работы программы:

```
1  stakenschneider@stakenschneider:~/temp$ ./p2.1 &
2  [1] 3133
3  stakenschneider@stakenschneider:~/temp$ jobs
4  [1]+  Running                 ./p2.1 &
5  stakenschneider@stakenschneider:~/temp$ kill -sigusr1 %1
6  SIGUSR catched.
7  stakenschneider@stakenschneider:~/temp$ jobs
8  [1]+  Running                 ./p2.1 &
9  stakenschneider@stakenschneider:~/temp$ kill -sigint %1
10 stakenschneider@stakenschneider:~/temp$ jobs
11 [1]+  Running                 ./p2.1 &
12 stakenschneider@stakenschneider:~/temp$ jobs
13 [1]+  Running                 ./p2.1 &
14 stakenschneider@stakenschneider:~/temp$ jobs
15 [1]+  Interrupt               ./p2.1
```

После отправки сигнала *SIGUSR1* вывелось сообщение и началась десятисекундная задержка. Далее был отправлен сигнал прерывания *SIGINT*, который был отправлен в очередь до ожидания завершения десятисекундной задержки.

2. Изменить обработчик так, чтобы отправка сигнала SIGINT производилась из обработчика функцией kill.

Изменим программу таким образом, чтобы отправка сигнала SIGINT производилась из обработчика функцией kill:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <signal.h>
4  #include <sys/types.h>
5  #include <sys/stat.h>
6  #include <fcntl.h>
7  #include <unistd.h>
8
9  // Задержка перед следующим сигналом
10 #define DELAY 10
11
12 // Функция, изменяющая набор сигналов
13 void (*specialSignal(int sig, void (*handler) (int))) (int);
14 // Обработчик сигнала
15 void userHandler(int sig);
16
17 int main() {
18   specialSignal(SIGUSR1, userHandler);
19
20   while(1)
21     pause();
22
23   return 0;
24 }
25
26 void (*specialSignal(int sig, void (*handler) (int))) (int) {
27   struct sigaction action;
```

```
28    // Задаем обработчик сигнала
29    action.sa_handler = handler;
30    // Инициализируем набор сигналов
31    sigemptyset(&action.sa_mask);
32    // Добавляем сигнал прерывания SIGINT в набор
33    sigaddset(&action.sa_mask, SIGINT);
34    // Инициализируем флаг нулем
35    action.sa_flags = 0;
36    // Изменим действие процесса при получении сигнала прерывания
37    if(sigaction(sig, &action, 0) < 0)
38      return SIG_ERR;
39
40    return action.sa_handler;
41  }
42
43  void userHandler(int sig) {
44    // Обрабатываем прерывание
45
46    // Если не тот сигнал
47    if(sig != SIGUSR1) {
48      perror("Wrong type of signal.\n");
49      return;
50    }
51
52    printf("SIGUSR catched.\n");
53
54    // Отправляем сигнал прерывания
55    kill(getpid(), SIGINT);
56
57    // Устанавливаем задержку перед обработкой следующего сигнала
58    sleep(DELAY);
59  }
```

Результат работы программы:

```
1  stakenschneider@stakenschneider$ gcc p2.2.c -o p2.2
2  stakenschneider@stakenschneider$ ./p2.2 &
3  [1] 3202
4  stakenschneider@stakenschneider$ kill -sigusr1 %1
5  SIGUSR catched.
6  stakenschneider@stakenschneider$ jobs
7  [1]+  Running                 ./p2.2 &
8  stakenschneider@stakenschneider$ jobs
9  [1]+  Running                  ./p2.2 &
10 stakenschneider@stakenschneider$ jobs
11 [1]+  Running                 ./p2.2 &
12 stakenschneider@stakenschneider$ jobs
13 [1]+  Interrupt               ./p2.2
```

Как и в предыдущем случае *SIGINT* попадает в очередь и ожидает завершения текущего обработчика.

3. Сравните обработчики надежных и ненадежных сигналов.

Отличие надежных сигналов от ненадежных состоит в том, что обработчик ненадежного сигнала срабатывает только один раз, после чего его нужно заново перезапустить внутри самого обработчика. Если в этот момент придет новый сигнал, то он будет потерян, т.к. обработчик не перезапустился и сигнал не обработался. В то время как в надежных сигналах нет этой проблемы, и они могут быть обработаны в любой момент времени.

1.3.3 Глава 3. Сигналы POSIX реального времени

В сигналах, стандартизированных в POSIX, существуют свои, отчасти досадные, ограничения. Например, нельзя быть уверенным в том, что множество сигналов, посланных поочередно, не будут слиты вместе.

Для обхода этого и других ограничений были созданы сигналы реального времени. Системы, которые поддерживают сигналы реального времени, поддерживают и стандартный механизм сигналов POSIX. Следует учитывать, что для достижения максимальной переносимости желательно применять POSIX-реализацию

сигналов, используя сигналы реального времени лишь там, где возникает необходимость в их особых свойствах.

Точные номера сигналов реального времени не специфицированы, но можно быть уверенным в том, что все сигналы с номерами между SIGRTMIN и SIGRTMAX являются сигналами реального времени. Для работы с сигналами реального времени применяются те же самые системные вызовы, что и для работы с обычными сигналами.

1-3. Провести эксперимент, позволяющий определить возможность организации очереди для различных типов сигналов, обычных и реального времени. Экспериментально подтвердить, что обработка равноприоритетных сигналов реального времени происходит в порядке очереди. Опытным путем определить наличие приоритетов сигналов реального времени.

Была разработана программа, которая организует очередь из определенного набора сигналов, которые могут прийти. Далее программа отправляет сама себе набор случайных сигналов из этого набора функцией *raise*:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <unistd.h>
#include <sys/signal.h>

// Количество возможных сигналов
#define COUNT_OF_SIGNALS 8
// Количество посылаемых случайных сиглаов
#define COUNT_OF_RANDOM_SIGNALS 20
// Ожидание приема сигналов
#define DELAY 1

// Задаем явно, тк.. стандартный SIGRTMIN не константа
#undef SIGRTMIN
#define SIGRTMIN 34

// Набор доступных сигналов для отправки
static int intSignal[COUNT_OF_SIGNALS] = {SIGUSR1, SIGUSR2, SIGCHLD, SIGRTMIN, SIGRTMIN + 1, SIGRTMIN + 2, SIGRTMIN + 3, SIGRTMIN + 4};

// Сигналы в виде строки для отображения
static char* stringSignal[COUNT_OF_SIGNALS] = {"SIGUSR1", "SIGUSR2", "SIGCHLD", "SIGRTMIN", "SIGRTMIN+1", "SIGRTMIN+2", "SIGRTMIN+3", "SIGRTMIN+4"};

const char* signalToString(int sig);
void signalHandler(int sig);

int main() {
  sigset_t mask;
  struct sigaction action;

  // Инициализация структур
  memset(&action, 0, sizeof(action));
  sigemptyset(&mask);

  // Добавляем в маску все возможные сигналы
  for(int index = 0; index < COUNT_OF_SIGNALS; ++index)
    sigaddset(&mask, intSignal[index]);

  // Устанавливаем обработчик для сигнала
  action.sa_handler = signalHandler;
  action.sa_mask = mask;

  // Изменение действия при получении сигнала
  for(int index = 0; index < COUNT_OF_RANDOM_SIGNALS; ++index)
    sigaction(intSignal[index], &action, NULL);

  // Перед отправкой сигналов блокируем обработчик сигналов
```

```
49    sigprocmask(SIG_BLOCK, &mask, NULL);
50
51    // Для правильной генерации чисел
52    srand(time(NULL));
53
54    printf("Send signals: ");
55    for(int index = 0; index < COUNT_OF_RANDOM_SIGNALS; ++index) {
56      // Случайный индекс массива всех возможных сигналов
57      int randomIndex = rand() % COUNT_OF_SIGNALS;
58      int currentSignal = intSignal[randomIndex];
59      // Отправка сигналов самому себе
60      raise(currentSignal);
61      printf("%s, ", signalToString(currentSignal));
62    }
63    printf("\nReceived signals: ");
64
65    // После отправки сигналов разблокируем обработчик
66    sigprocmask(SIG_UNBLOCK, &mask, NULL);
67
68    // Ожидаем завершение обработки сигналов
69    sleep(DELAY);
70
71    printf("\n");
72    return 0x0;
73  }
74
75  void signalHandler(int sig) {
76    printf("%s, ", signalToString(sig));
77  }
78
79  const char* signalToString(int sig) {
80    int contains = -1;
81    for(int index = 0; index < COUNT_OF_SIGNALS; ++index)
82      if(intSignal[index] == sig) {
83        contains = index;
84        break;
85      }
86
87    // Если сигнал не содержится в массиве доступных сигналов, NULL
88    if(contains == -1)
89      return NULL;
90
91    // Если содержится, то возвращаем строку
92    return stringSignal[contains];
93  }
```

Результат работы программы:

```
1  stakenschneider@stakenschneider:~/temp$ gcc p3.1.c -o p3.1
2
3  stakenschneider@stakenschneider:~/temp$ ./p3.1
4  Send signals: SIGRTMIN, SIGCHLD, SIGCHLD, SIGRTMIN, SIGRTMIN, SIGRTMIN, SIGCHLD, SIGRTMIN
       +4, SIGRTMIN+4, SIGUSR1, SIGUSR2, SIGUSR1, SIGUSR2, SIGRTMIN+2, SIGRTMIN+2, SIGCHLD,
       SIGRTMIN+2, SIGRTMIN+1, SIGUSR1, SIGRTMIN+2,
5  Received signals: SIGUSR1, SIGUSR2, SIGCHLD, SIGRTMIN, SIGRTMIN, SIGRTMIN, SIGRTMIN,
       SIGRTMIN+1, SIGRTMIN+2, SIGRTMIN+2, SIGRTMIN+2, SIGRTMIN+2, SIGRTMIN+4, SIGRTMIN+4,
6
7  stakenschneider@stakenschneider:~/temp$ ./p2.3
8  Send signals: SIGRTMIN+1, SIGRTMIN+3, SIGRTMIN+4, SIGUSR2, SIGRTMIN, SIGUSR1, SIGRTMIN+1,
       SIGRTMIN+2, SIGRTMIN+4, SIGUSR1, SIGRTMIN+2, SIGRTMIN+2, SIGRTMIN+4, SIGRTMIN,
       SIGRTMIN, SIGUSR1, SIGRTMIN+1, SIGRTMIN+1, SIGRTMIN+1, SIGRTMIN+1,
9  Received signals: SIGUSR1, SIGUSR2, SIGRTMIN, SIGRTMIN, SIGRTMIN, SIGRTMIN+1, SIGRTMIN+1,
       SIGRTMIN+1, SIGRTMIN+1, SIGRTMIN+1, SIGRTMIN+1, SIGRTMIN+2, SIGRTMIN+2, SIGRTMIN+2,
       SIGRTMIN+3, SIGRTMIN+4, SIGRTMIN+4, SIGRTMIN+4,
```

Из результатов программы хорошо видно, что множества посланных и принятых сигналов отличаются, это связано с тем что обычные сигналы слились в один. Все сигналы были отправлены одновременно, однако

они приходят в порядке очереди (по коду сигнала).

1.3.4 Глава 4. Неименованные каналы

Неименованный канал - это один из методов межпроцессного взаимодействия (*IPC*) в операционной системе, который доступен связанным процессам — родительскому и дочернему. Представляется в виде области памяти на внешнем запоминающем устройстве, управляемой операционной системой, которая осуществляет выделение взаимодействующим процессам частей из этой области памяти для совместной работы. Организация данных в канале использует стратегию *FIFO*, то есть информация, которая первой записана в канал, будет первой прочитана из канала.

Важное отличие неименованного канала от файла заключается в том, что прочитанная информация немедленно удаляется из него и не может быть прочитана повторно.

1. Организовать программу так, чтобы процесс-родитель создавал неименованный канал, создавал потомка, закрывал канал на запись и записывал в произвольный текстовый файл считываемую информацию. В процессе-потомке будет происходить считывание данных из файла и запись в канал.

Разработаем программу, в которой процесс-родитель создает неименованный канал и потомка, закрывает канал на запись и записывает полученный из канала текст в текстовый файл. Процесс-потомок производит чтение из файла и запись в файл:

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

// Название файла для чтения
#define READ_FILE "readfile.txt"
// Название файла для записи
#define WRITE_FILE "writefile.txt"
// Размер буфера для считывания и записи
#define BUFFER_SIZE 100

int main() {
  // Пара дескрипторов канала
  int pipeArray[2];

  // Пробуем создать неименованный канал
  if(pipe(pipeArray) < 0) {
    perror("It's impossible to create pipe.\n");
    return 0x1;
  }

  int result;

  if(!fork()) {
    // Закрываем канал на чтение
    close(pipeArray[0]);

    // Пробуем открыть файл на чтение
    FILE* file = fopen(READ_FILE, "r");
    if(!file) {
      perror("Child couldn't open file to read.\n");
      return 0x1;
    }

    char buffer[BUFFER_SIZE];
    while(!feof(file)) {
      // Считываем файл по размеру буффера
      result = fread(buffer, sizeof(char), BUFFER_SIZE, file);
      // Записываем результат чтения в неименованный канал
      write(pipeArray[1], buffer, result);
    }

```

```
    // Закрываем файл
    fclose(file);
    // Закрываем канал на запись
    close(pipeArray[1]);
    return 0x0;
  } else {
    // Закрываем канал на запись
    close(pipeArray[1]);

    // Пробуем открыть файл на запись
    FILE* file = fopen(WRITE_FILE, "w");
    if(!file) {
      perror("Parent couldn't open file to write.\n");
      return 0x2;
    }

    printf("Read from pipe:\n");

    char buffer[BUFFER_SIZE];
    while(1) {
      // Очищаем буфер
      bzero(buffer, BUFFER_SIZE);
      // Считываем данные из канала в буфер
      result = read(pipeArray[0], buffer, BUFFER_SIZE);

      if(!result)
        break;

      printf("%s", buffer);
      // Записываем считанные данные в выходной файл
      fwrite(buffer, sizeof(char), result, file);
    }

    // Закрываем файл
    fclose(file);
    // Закрываем канал на чтение
    close(pipeArray[0]);
    return 0x0;
  }
}
```

Содержимое файла для чтения:

```
some text there
AAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBB
CCCCCCCCCCCCCCCCCC
```

Результат работы программы:

```
stakenschneider@stakenschneider:~/temp$ ./p4.1
Read from pipe:
some text there
AAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBB
CCCCCCCCCCCCCCCCCC
```

Содержимое файла записи после выполнения программы:

```
some text there
AAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBB
CCCCCCCCCCCCCCCCCC
```

Программа успешно создала неименованный канал, который в одностороннем порядке передал данные от процесса-потомка процессу-родителю.

Главное применение неименованных каналов в ОС UNIX - реализация конвейеров команд в интерпретаторах командной строки.

1.3.5 Глава 5. Именованные каналы

Именованные каналы во многом работают так же, как и обычные каналы, но все же имеют несколько заметных отличий.

- Именованные каналы существуют в виде специального файла устройства в файловой системе.
- Процессы различного происхождения могут разделять данные через такой канал.
- Именованный канал остается в файловой системе для дальнейшего использования и после того, как весь ввод/вывод сделан.

1. Создать клиент-серверное приложение, демонстрирующее дуплексную передачу информации двумя однонаправленными именованными каналами между клиентом и сервером.

Разработаем программу-сервер, создающую два именованных канала, используя функцию *mknod*. Для установки прав для доступа на чтение и запись используем флаг права доступа с маской *S_FIFO | 0666*. Первый канал открывается на запись, а второй на чтение. Сервер передает клиенту имя файла для чтения и ожидает содержимое файла от клиента:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define DEFAULT_FILENAME "readfile.txt"
#define BUFFER_SIZE 100

int main(int argc, char** argv) {
  char* filename = DEFAULT_FILENAME;

  if(argc < 2)
    printf("Using default filename: %s.\n", filename);
  else
    strcpy(filename, argv[1]);

  // Создаем первый канал
  int result = mknod("firstChannel", S_IFIFO | 0666, 0);
  if(result) {
    perror("It's impossible to create first channel.\n");
    return 0x1;
  }

  // Создаем второй канал
  result = mknod("secondChannel", S_IFIFO | 0666, 0);
  if(result) {
    perror("It's impossible to create second channel.\n");
    return 0x2;
  }

  // Открываем первый канал на запись
  int firstChannel = open("firstChannel", O_WRONLY);
  if(firstChannel == -1) {
    perror("It's impossible to open first channel for writing.\n");
    return 0x3;
  }

  // Открываем первый канал на чтение
  int secondChannel = open("secondChannel", O_RDONLY);
  if(secondChannel == -1) {
    perror("It's impossible to open second channel for reading.\n");
    return 0x4;
  }

```

```
48    // Записываем в первый канал имя файла
49    write(firstChannel, filename, strlen(filename));
50
51    char buffer[BUFFER_SIZE];
52    while(1) {
53      bzero(buffer, BUFFER_SIZE);
54      // Считываем из второго канала и выводим содержимое
55      result = read(secondChannel, buffer, BUFFER_SIZE);
56      if(result <= 0)
57        break;
58      printf("File's part: %s\n", buffer);
59    }
60
61    // Закрываем каналы
62    close(firstChannel);
63    close(secondChannel);
64
65    // Разрываем связь и удаляем файлы
66    unlink("firstChannel");
67    unlink("secondChannel");
68
69    return 0x0;
70  }
```

Программа клиент, в свою очередь, симметрично серверу открывает первый канал на чтение, а второй на запись. Клиент принимает имя файла для чтения и передает серверу его содержимое:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <sys/types.h>
6  #include <sys/stat.h>
7  #include <fcntl.h>
8
9  #define BUFFER_SIZE 100
10
11 int main() {
12   // Каналы уже созданы сервером, открываем каналы
13   int firstChannel = open("firstChannel", O_RDONLY);
14   if(firstChannel == -1) {
15     perror("It's impossible to open first channel for reading.\n");
16     return 0x1;
17   }
18
19   int secondChannel = open("secondChannel", O_WRONLY);
20   if(secondChannel == -1) {
21     perror("It's impossible to open second channel for writing.\n");
22     return 0x2;
23   }
24
25   char filename[BUFFER_SIZE];
26   bzero(filename, BUFFER_SIZE);
27
28   // Считываем из первого канала имя файла
29   int result = read(firstChannel, filename, BUFFER_SIZE);
30   if(result <= 0) {
31     perror("It's impossible to read filename from first channel.\n");
32     return 0x3;
33   }
34
35   printf("Received filename: %s.\n", filename);
36
37   // Открываем файл на чтение
38   FILE* file = fopen(filename, "r");
39   if(!file) {
40     perror("It's impossible open file to read.\n");
```

```
41      return 0x4;
42    }
43
44    // Отправляем во второй канал содержимое файла
45    char* buffer[BUFFER_SIZE];
46    while(!feof(file)) {
47      result = fread(buffer, sizeof(char), BUFFER_SIZE, file);
48      write(secondChannel, buffer, result);
49    }
50
51    // Закрываем файл
52    fclose(file);
53
54    // Закрываем каналы
55    close(firstChannel);
56    close(secondChannel);
57    return 0x0;
58  }
```

Результат работы программ:

```
stakenschneider@stakenschneider:~/temp$ gcc p5.1.se.c -o p5.1.se
stakenschneider@stakenschneider:~/temp$ gcc p5.1.cl.c -o p5.1.cl
stakenschneider@stakenschneider:~/temp$ ./p5.1.se &
[1] 5778
stakenschneider@stakenschneider:~/temp$ Using default filename: readfile.txt.
./p5.1.cl
Received filename: readfile.txt.
File's part: some text there
AAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBB
CCCCCCCCCCCCCCCCCC

stakenschneider@stakenschneider:~/temp$ jobs
[1]+  Done                    ./p5.1.se
```

В фоновом режиме был запущен сервер и клиент. Сервер успешно передал клиенту имя файла, в свою очередь клиент успешно отправил на сервер его содержимое.

В процессе работы программ были созданы два файла:

```
stakenschneider@stakenschneider:~/temp$ ls -l
total 248
(...)
prw-rw-r-- 1 stakenschneider stakenschneider     0 дек 11 22:52 firstChannel
(...)
prw-rw-r-- 1 stakenschneider stakenschneider     0 дек 11 22:52 secondChannel
(...)
```

Файлы имеют тип *p*, что означает, что это файлы именованного канала. Несмотря на записанные данные, размер этих файлов равен нулю. Это означает, что файл используется не как хранилище пересылаемых данных, а только для получения информации системой о них. Сами данные проходят через ядро ОС. Это делает невозможным использование каналов в межсетевом общении.

1.3.6 Глава 6. Очереди сообщений

Очереди сообщений представляют собой связный список в адресном пространстве ядра. Сообщения могут посылаться в очередь по порядку и доставаться из очереди несколькими разными путями. Каждая очередь сообщений однозначно определена идентификатором *IPC*.

В общих чертах обмен сообщениями выглядит примерно так: один процесс помещает сообщение в очередь посредством неких системных вызовов, а любой другой процесс может прочитать его оттуда, при условии, что и процесс-источник сообщения и процесс-приемник сообщения используют один и тот же ключ для получения доступа к очереди.

Для определения некоторых констант, связанных с *IPC*, воспользуемся командой *ipcs* с ключем *-l*:

```
stakenschneider@stakenschneider:~/temp$ ipcs -l

```

```
------ Messages Limits --------
max queues system wide = 32000
max size of message (bytes) = 8192
default max size of queue (bytes) = 16384

------ Shared Memory Limits --------
max number of segments = 4096
max seg size (kbytes) = 18014398509465599
max total shared memory (kbytes) = 18014398442373116
min seg size (bytes) = 1

------ Semaphore Limits --------
max number of arrays = 32000
max semaphores per array = 32000
max semaphores system wide = 1024000000
max ops per semop call = 500
semaphore max value = 32767
```

1. Создать клиент-серверное приложение, демонстрирующее передачу информации между процессами посредством очереди сообщений.

Разработаем программу-сервер, которая создает очередь, используя функцию *msgget* и ожидает прием сообщений от клиентов. Как только приходит сообщение от клиента, сразу же отсылается ответное сообщение с другим типом посылки. Обработчик сигнала прерывания необходим для корректного завершения очереди:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/msg.h>
#include <signal.h>

#define DEFAULT_FILENAME "readfile.txt"
#define BUFFER_SIZE 100

// Тип присылаемого пакета
#define MESSAGE_TYPE 1L

// Текст ответа
#define RESPONSE_STRING "Message received!"
// Тип посылаемого ответа
#define RESPONSE_TYPE 2L

// Структура сообщения
typedef struct {
  // Тип сообщения
  long type;
  // Текст сообщения
  char buffer[BUFFER_SIZE];
} Message;

// Глобальная переменная для доступа к очереди
int queue;

// Обработчик сигнала прерывания с терминала для корректного удаления очереди
void signalHandler(int sig);

int main(int argc, char** argv) {
  char* filename = DEFAULT_FILENAME;

  if(argc < 2)
    printf("Using default filename: %s.\n", filename);
  else
```

```
42      strcpy(filename, argv[1]);
43
44    // Пробуем создать ключ
45    key_t key = ftok(filename, 'Q');
46    if(key == -1) {
47      perror("It's impossible to get key for key file.\n");
48      return 0x1;
49    }
50
51    // Получаем идентификатор очереди
52    queue = msgget(key, IPC_CREAT | 0666);
53    if(queue < 0) {
54      perror("It's impossible to create queue.\n");
55      return 0x2;
56    }
57
58    // Обработка сигнала прерывания с терминала для корректного удаления очереди
59    signal(SIGINT, signalHandler);
60
61    Message message;
62
63    while(1) {
64      bzero(message.buffer, BUFFER_SIZE);
65
66      // Принимаем сообщения определенного типа от клиентов
67      int result = msgrcv(queue, &message, sizeof(Message), MESSAGE_TYPE, 0);
68      if(result < 0) {
69        perror("It's impossible to receive message.\n");
70        signalHandler(SIGINT);
71      }
72      perror("Client's message: %s", message.buffer);
73
74      // Формируем ответ
75      message.type = RESPONSE_TYPE;
76      bzero(message.buffer, BUFFER_SIZE);
77      strcpy(message.buffer, RESPONSE_STRING);
78
79      // Отправляем ответ
80      result = msgsnd(queue, (void*) &message, sizeof(Message), 0);
81      if(result != 0) {
82        perror("It's impossible to send message.\n");
83        signalHandler(SIGINT);
84      }
85    }
86
87    return 0x0;
88  }
89
90  void signalHandler(int sig) {
91    // Возвращаем обработчик сигнала по умолчанию
92    signal(sig, SIG_DFL);
93
94    // Удаляем очередь
95    if(msgctl(queue, IPC_RMID, 0) < 0)
96      perror("It's impossible to delete queue.\n");
97    else
98      perror("Queue has been successfully deleted.\n");
99
100   exit(0x3);
101 }
```

Программа-клиент симметрична серверной программе: она получает строку с консоли, отсылает ее серверу и получает ответ:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
```

```c
#include <string.h>
#include <sys/ipc.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <signal.h>

#define DEFAULT_FILENAME "readfile.txt"
#define BUFFER_SIZE 100

// Тип посылаемого пакета
#define MESSAGE_TYPE 1L
// Тип присылаемого ответа
#define RESPONSE_TYPE 2L

// Структура сообщения
typedef struct {
  // Тип сообщения
  long type;
  // Текст сообщения
  char buffer[BUFFER_SIZE];
} Message;

int main(int argc, char** argv) {
  char* filename = DEFAULT_FILENAME;

  if(argc < 2)
    printf("Using default filename: %s.\n", filename);
  else
    strcpy(filename, argv[1]);

  // Пробуем создать ключ
  key_t key = ftok(filename, 'Q');
  if(key == -1) {
    perror("It's impossible to get key for key file.\n");
    return 0x1;
  }

  // Получаем идентификатор очереди
  int queue = msgget(key, 0);
  if(queue < 0) {
    perror("It's impossible to create queue.\n");
    return 0x2;
  }

  Message message;

  while(1) {
    // Формируем сообщение

    // Считываем сообщение из потока ввода
    bzero(message.buffer, BUFFER_SIZE);
    fgets(message.buffer, BUFFER_SIZE, stdin);

    // Задаем тип сообщения
    message.type = MESSAGE_TYPE;

    // Отправка сообщения
    int result = msgsnd(queue, (void*) &message, sizeof(Message), 0);
    if(result != 0) {
      perror("It's impossible to send message.\n");
      return 0x3;
    }

    // Получение ответа
    result = msgrcv(queue, &message, sizeof(Message), RESPONSE_TYPE, 0);
    if(result < 0) {
```

```
70        perror("It's impossible to receive message.\n");
71        return 0x4;
72      }
73
74      printf("Server's message: %s\n", message.buffer);
75    }
76
77    return 0x0;
78  }
```

Результат выполнения программ:

```
1  # Terminal 1
2  stakenschneider@stakenschneider:~/temp$ gcc p6.1.se.c -o p6.1.se
3  stakenschneider@stakenschneider:~/temp$ gcc p6.1.cl.c -o p6.1.cl
4  stakenschneider@stakenschneider:~/temp$ ./p6.1.se
5  Using default filename: readfile.txt.
6  Client's message: 11111111111
7  Client's message: 22222222222
8  Client's message: 33333333333
9  Client's message: 44444444444
10 ^CQueue has been successfully deleted.
11
12 # Terminal 2
13 stakenschneider@stakenschneider:~/temp$ ./p6.1.cl
14 Using default filename: readfile.txt.
15 11111111111
16 Server's message: Message received!
17 33333333333
18 Server's message: Message received!
19 12345_SERVER_IS_DOWN_ALREADY_12345
20 It's impossible to send message.
21
22 # Terminal 3
23 stakenschneider@stakenschneider:~/temp$ ./p6.1.cl
24 Using default filename: readfile.txt.
25 22222222222
26 Server's message: Message received!
27 44444444444
28 Server's message: Message received!
29 12345_SERVER_IS_DOWN_ALREADY_12345
30 It's impossible to send message.
```

В этом эксперименте был запущен сервер, который успешно принимал сообщения от двух клиентов. После этого сервер был корректно отключен прерыванием с терминала, вызвался обработчик прерывания и очередь успешно удалилась. Клиенты узнали о недоступности сервера только после отправки последнего сообщения.

Команда ipcs выводит информацию о текущем состоянии IPC операционной системы. Воспользуемся ей для того, чтобы определить, находится ли только что созданная очередь в списке очередей:

```
1  stakenschneider@stakenschneider:~/temp$ ipcs—
2
3  ——— Message Queues ———————
4  key msqid owner perms used-bytes messages
5  0x51010071 0 stakenschneider 666 0 0—
6
7  ——— Shared Memory Segments ———————
8  key shmid owner perms bytes nattch status
9  0x00000000 557056 stakenschneider 600 524288 2 dest
10 0x00000000 163841 stakenschneider 600 524288 2 dest
11 0x00000000 589826 stakenschneider 600 2097152 2 dest
12 0x00000000 688131 stakenschneider 600 524288 2 dest
13 0x00000000 458756 stakenschneider 600 16777216 2
14 0x00000000 1114117 stakenschneider 600 524288 2 dest
15 0x00000000 524294 stakenschneider 600 524288 2 dest
16 0x00000000 786439 stakenschneider 600 524288 2 dest
17 0x00000000 1212424 stakenschneider 600 524288 2 dest
18 0x00000000 983049 stakenschneider 600 524288 2 dest
```

```
0x00000000 1015818 stakenschneider 600 33554432 2 dest
0x00000000 1310731 stakenschneider 600 524288 2 dest—

—— Semaphore Arrays ——————
key semid owner perms nsems
```

И действительно, в списке "Message Queues"появилась только что созданная очередь.

1.3.7 Глава 7. Семафоры и разделяемая память

Семафор — самый часто употребляемый метод для синхронизации потоков и для контролирования одновременного доступа множеством потоков/процессов к общей памяти (к примеру, глобальной переменной). Взаимодействие между процессами в случае с семафорами заключается в том, что процессы работают с одним и тем же набором данных и корректируют свое поведение в зависимости от этих данных.

1. Есть один процесс, выполняющий запись в разделяемую память и один процесс, выполняющий чтение из нее. Под чтением понимается извлечение данных из памяти. Программа должна обеспечить невозможность повторного чтения одних и тех же данных и невозможность перезаписи данных, т.е. новой записи, до тех пор, пока читатель не прочитает предыдущую.

Для реализации программы используем два бинарных семафора (ожидание освобождения ресурса - 1, последующий захват ресурса - 0), освобождение ресурса это также установка семафора в 1. Пару таких семафоров иногда называют разделенным бинарным семафором, поскольку в любой момент времени только один из них может иметь значение 1. При таком алгоритме работы оба процесса после выполнения своей задачи и освобождения одного из семафоров будут ждать освобождения другого семафора, которое произведет другой процесс, но только после выполнения своей работы. Таким образом повторное чтение или повторная запись стала невозможной.

В качестве общей памяти была использована структура *Message* из предыдущего пункта.

Обработчик сигнала прерывания необходим для корректного завершения работы с семафорами.

Программа-сервер:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/time.h>

#define DEFAULT_FILENAME "readfile.txt"
#define BUFFER_SIZE 100

// Структура сообщения
typedef struct {
  // Тип сообщения
  long type;
  // Текст сообщения
  char buffer[BUFFER_SIZE];
} Message;

static struct sembuf LOCK[1] = {0, -1, 0};
static struct sembuf RELEASE[1] = {1, 1, 0};

Message* message;
int semaphore;
int shmemory;

// Обработчик сигнала прерывания необходим( для корректного завершения работы при прерывании)
void signalHandler(int sig);
// Функция для корректного завершения работы приложения
void clearAndExit(int code);

int main(int argc, char** argv) {
```

```
35    char* filename = DEFAULT_FILENAME;
36
37    if(argc < 2)
38      printf("Using default filename: %s.\n", filename);
39    else
40      strcpy(filename, argv[1]);
41
42    // Пробуем создать ключ
43    key_t key = ftok(filename, 'Q');
44    if(key == -1) {
45      perror("It's impossible to get key for key file.\n");
46      return 0x1;
47    }
48
49    // Создаем shm
50    shmemory = shmget(key, sizeof(Message), IPC_CREAT | 0666);
51    if(shmemory < 0) {
52      perror("It's impossible to create shm.\n");
53      return 0x2;
54    }
55
56    // Присоединяем shm в наше адресное пространство
57    message = (Message*) shmat(shmemory, 0, 0);
58    if(message < 0) {
59      perror("It's impossible to attach shm.\n");
60      return 0x3;
61    }
62
63    // Обработка сигнала прерывания с терминала для корректного завершения работы
64    signal(SIGINT, signalHandler);
65
66    // Создание группы из двух семафоров: первый показывает, что можно читать, второй — что можно
        писать
67    semaphore = semget(key, 2, IPC_CREAT | 0666);
68    if(semaphore < 0) {
69      perror("It's impossible to create semaphore.\n");
70      clearAndExit(0x4);
71    }
72
73    // Устанавливаем второй семофор в единицу можно( писать)
74    int result = semop(semaphore, SET_WRITE_ENABLE, 1);
75    if(result < 0) {
76      perror("It's imposibble to set write enable.\n");
77      clearAndExit(0x5);
78    }
79
80    while(1) {
81      // Ожидаем начало работы клиента
82      result = semop(semaphore, LOCK, 1);
83      if(result < 0) {
84        perror("It's imposibble to lock.\n");
85        clearAndExit(0x6);
86      }
87
88      printf("Client's message: %s", message->buffer);
89
90      // Устанавливаем второй семофор в единицу можно( писать)
91      result = semop(semaphore, RELEASE, 1);
92      if(result < 0) {
93        perror("It's imposibble to release.\n");
94        clearAndExit(0x7);
95      }
96    }
97
98    return 0x0;
99  }
```

```
100
101 void signalHandler(int sig) {
102   // Корректно завершаем работу приложения
103   clearAndExit(0x8);
104 }
105
106 void clearAndExit(int code) {
107   // Устанавливаем обработчик сигнала по умолчанию
108   signal(SIGINT, SIG_DFL);
109
110   // Отключаем разделяемую память
111   int result = shmdt(message);
112   if(result < 0)
113     perror("It's impossible to detach shm.\n");
114
115   // Удаление shm
116   result = shmctl(shmemory, IPC_RMID, 0);
117   if(result < 0)
118     perror("It's impossible to delete shm.\n");
119
120   // Удаление семафоров
121   result = semctl(semaphore, 0, IPC_RMID);
122   if(result < 0)
123     perror("It's impossible to delete semaphore.\n");
124
125   printf("Clear finished.\n");
126   exit(code);
127 }
```

Программа-клиент:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <signal.h>
6 #include <sys/ipc.h>
7 #include <sys/sem.h>
8 #include <sys/shm.h>
9 #include <sys/time.h>
10
11 #define DEFAULT_FILENAME "readfile.txt"
12 #define BUFFER_SIZE 100
13
14 // Структура сообщения
15 typedef struct {
16   // Тип сообщения
17   long type;
18   // Текст сообщения
19   char buffer[BUFFER_SIZE];
20 } Message;
21
22 static struct sembuf WAIT[1] = {1, -1, 0};
23 static struct sembuf UNLOCK[1] = {0, 1, 0};
24
25 Message* message;
26 int semaphore;
27 int shmemory;
28
29 // Обработчик сигнала прерывания необходим( для корректного завершения работы при прерывании)
30 void signalHandler(int sig);
31 // Функция для корректного завершения работы приложения
32 void clearAndExit(int code);
33
34 int main(int argc, char** argv) {
35   char* filename = DEFAULT_FILENAME;
36
```

```
37    if(argc < 2)
38      printf("Using default filename: %s.\n", filename);
39    else
40      strcpy(filename, argv[1]);
41
42    // Пробуем создать ключ
43    key_t key = ftok(filename, 'Q');
44    if(key == -1) {
45      perror("It's impossible to get key for key file.\n");
46      return 0x1;
47    }
48
49    // Создаем shm
50    shmemory = shmget(key, sizeof(Message), 0666);
51    if(shmemory < 0) {
52      perror("It's impossible to create shm.\n");
53      return 0x2;
54    }
55
56    // Присоединяем shm в наше адресное пространство
57    message = (Message*) shmat(shmemory, 0, 0);
58    if(message < 0) {
59      perror("It's impossible to attach shm.\n");
60      return 0x3;
61    }
62
63    // Обработка сигнала прерывания с терминала для корректного завершения работы
64    signal(SIGINT, signalHandler);
65
66    // Создание группы из двух семафоров: первый показывает, что можно читать, второй — что можно
        писать
67    semaphore = semget(key, 2, 0666);
68    if(semaphore < 0) {
69      perror("It's impossible to create semaphore.\n");
70      clearAndExit(0x4);
71    }
72
73    char buffer[BUFFER_SIZE];
74    while(1) {
75      // Получаем сообщение из входного потока
76      bzero(buffer, BUFFER_SIZE);
77      fgets(buffer, BUFFER_SIZE, stdin);
78
79      // Настройка для отправки сообщения
80      int result = semop(semaphore, WAIT, 1);
81      if(result < 0) {
82        perror("It's imposibble to write.\n");
83        clearAndExit(0x5);
84      }
85
86      // Записываем сообщение в разделяемую память
87      sprintf(message->buffer, "%s", buffer);
88
89      // Отправляем серверу сообщение что можно писать
90      result = semop(semaphore, UNLOCK, 1);
91      if(result < 0) {
92        perror("It's imposibble to set write enable.\n");
93        clearAndExit(0x6);
94      }
95    }
96
97    return 0x0;
98  }
99
100 void signalHandler(int sig) {
101   // Корректно завершаем работу приложения
```

```
102   clearAndExit(0x7);
103 }
104
105 void clearAndExit(int code) {
106   // Устанавливаем обработчик сигнала по умолчанию
107   signal(SIGINT, SIG_DFL);
108
109   // Отключаем разделяемую память
110   int result = shmdt(message);
111   if(result < 0)
112     perror("It's impossible to detach shm.\n");
113
114   printf("Clear finished.\n");
115   exit(code);
116 }
```

Результаты работы программ:

```
1  # Terminal 1
2  stakenschneider@stakenschneider:~/temp$ gcc p7.1.1.se.c -o p7.1.1.se
3  stakenschneider@stakenschneider:~/temp$ gcc p7.1.1.cl.c -o p7.1.1.cl
4  stakenschneider@stakenschneider:~/temp$ ./p7.1.1.se
5  Using default filename: readfile.txt.
6  Client's message: 111111111
7  Client's message: 333333333
8  Client's message: 222222222
9  Client's message: 444444444
10 ^CClear finished.
11
12 # Terminal 2
13 stakenschneider@stakenschneider:~/temp$ ./p7.1.1.cl
14 Using default filename: readfile.txt.
15 111111111
16 333333333
17 ^CClear finished.
18
19 # Terminal 3
20 stakenschneider@stakenschneider:~/temp$ ./p7.1.1.cl
21 Using default filename: readfile.txt.
22 222222222
23 444444444
24 ^CClear finished.
```

В этом эксперименте был запущен сервер, который успешно принимал сообщения от двух клиентов. После этого сервер был корректно отключен прерыванием с терминала, вызвался обработчик прерывания и семафоры успешно удалилась.

2. Добавление условия корректной работы нескольких читателей и нескольких писателей одновременно.

Добавление этого условия не приводит к необходимости использования дополнительных средств синхронизации. Теперь вместо одного процесса за каждый семафор будут конкурировать насколько процессов, но повторная запись и чтение все также невозможно, так как, чтобы очередной клиент работал нужно освобождение семафора, которое выполняется из процесса-читателя, и наоборот.

3. Добавление не единичного буфера, а буфера некоторого размера. Тип буфера не имеет значения.

Так как размер буфера не равен единице, то больше нет необходимости в чередовании операций чтения и записи, допустима ситуация нескольких записей подряд и после этого нескольких чтений. Нужно только следить, чтобы не было записи в уже заполненный буфер и не было чтения из пустого буфера.

Так как семафоры не бинарные, захватить их может сразу несколько процессов, то есть несколько процессов попадут в секцию записи или чтения. В этом случае, если операция записи или чтения не атомарная, может произойти нарушение нормальной работы программы, к примеру, несколько клиентов попытаются произвести запись в одну и ту же ячейку буфера. Таким образом, операции записи-чтения становятся критическими секциями, доступ к которым также необходимо синхронизировать. Для этого будет достаточно еще

одного бинарного семафора, имеющего смысл разрешения доступа к памяти. Оба типа процессов должны захватывать его при попытке взаимодействия с памятью и освобождать после.

Порядок операций освобождения семафоров не важен, в то же время изменение порядка захвата семафоров может привести к взаимной блокировке процессов (dead lock).

Программа-сервер:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/time.h>

#define DEFAULT_FILENAME "readfile.txt"
#define BUFFER_SIZE 100
#define ITERATIONS_COUNT 25

static struct sembuf WAIT[1] = {2, -1, 0};
static struct sembuf RELEASE[1] = {1, 1, 0};
static struct sembuf FREE[1] = {1, BUFFER_SIZE, 0};
static struct sembuf LOCK[1] = {0, -1, 0};
static struct sembuf UNLOCK[1] = {0, 1, 0};

int* buffer;
int shmemory;
int semaphore;

// Обработчик сигнала прерывания необходим( для корректного завершения работы при прерывании)
void signalHandler(int sig);
// Функция для корректного завершения работы приложения
void clearAndExit(int code);

int main(int argc, char** argv) {
  char* filename = DEFAULT_FILENAME;

  if(argc < 2)
    printf("Using default filename: %s.\n", filename);
  else
    strcpy(filename, argv[1]);

  // Пробуем создать ключ
  key_t key = ftok(filename, 'Q');
  if(key == -1) {
    perror("It's impossible to get key for key file.\n");
    return 0x1;
  }

  // Создаем shm
  shmemory = shmget(key, (BUFFER_SIZE + 1) * sizeof(int), IPC_CREAT | 0666);
  if(shmemory < 0) {
    perror("It's impossible to create shm.\n");
    return 0x2;
  }

  // Присоединяем shm в наше адресное пространство
  buffer = (int*) shmat(shmemory, 0, 0);
  if(buffer < 0) {
    perror("It's impossible to attach shm.\n");
    return 0x3;
  }

  // Обработка сигнала прерывания с терминала для корректного завершения работы
  signal(SIGINT, signalHandler);
```

```
61
62    // Создание группы из двух семафоров: первый показывает, что можно читать, второй — что можно
        писать
63    semaphore = semget(key, 3, IPC_CREAT | 0666);
64    if(semaphore < 0) {
65      perror("It's impossible to create semaphore.\n");
66      clearAndExit(0x4);
67    }
68
69    // Инициализация буфера
70    for(int index = 0; index <= BUFFER_SIZE; ++index)
71      buffer[index] = -1;
72
73
74    // Установка единицы в число свободных ячеек
75    int result = semop(semaphore, FREE, 1);
76    if(result < 0) {
77      perror("It's imposibble to set free.\n");
78      clearAndExit(0x5);
79    }
80
81    // Разблокирование
82    result = semop(semaphore, UNLOCK, 1);
83    if(result < 0) {
84      perror("It's imposibble to set unlock.\n");
85      clearAndExit(0x6);
86    }
87
88    printf("Press \"Enter\" to start.\n");
89    getchar();
90
91    for(int index = 0; index < ITERATIONS_COUNT; ++index) {
92      // Ожидаем хотя бы одну непустую ячейку
93      result = semop(semaphore, WAIT, 1);
94      if(result < 0) {
95        perror("It's imposibble to wait not empty.\n");
96        clearAndExit(0x7);
97      }
98
99      // Ожидаем возможность работы с памятью
100     result = semop(semaphore, LOCK, 1);
101     if(result < 0) {
102       perror("It's imposibble to set lock.\n");
103       clearAndExit(0x8);
104     }
105
106     // Считывание сообщения от клиента
107     result = buffer[buffer[BUFFER_SIZE]];
108     printf("Remove from cell #%d, value %d.\n", buffer[BUFFER_SIZE], result);
109     --buffer[BUFFER_SIZE];
110
111     // Разблокируем
112     result = semop(semaphore, UNLOCK, 1);
113     if(result < 0) {
114       perror("It's imposibble to set unlock.\n");
115       clearAndExit(0x9);
116     }
117
118     // Увеличение числа пустых ячеек
119     result = semop(semaphore, RELEASE, 1);
120     if(result < 0) {
121       perror("It's imposibble to release.\n");
122       clearAndExit(0xA);
123     }
124   }
125
```

```
126   return 0x0;
127 }
128
129 void signalHandler(int sig) {
130   // Корректно завершаем работу приложения
131   clearAndExit(0xB);
132 }
133
134 void clearAndExit(int code) {
135   // Устанавливаем обработчик сигнала по умолчанию
136   signal(SIGINT, SIG_DFL);
137
138   // Отключаем разделяемую память
139   int result = shmdt(buffer);
140   if(result < 0)
141     perror("It's impossible to detach shm.\n");
142
143   // Удаление shm
144   result = shmctl(shmemory, IPC_RMID, 0);
145   if(result < 0)
146     perror("It's impossible to delete shm.\n");
147
148   // Удаление семафоров
149   result = semctl(semaphore, 0, IPC_RMID);
150   if(result < 0)
151     perror("It's impossible to delete semaphore.\n");
152
153   printf("Clear finished.\n");
154   exit(code);
155 }
```

Программа-клиент:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <signal.h>
6  #include <sys/ipc.h>
7  #include <sys/sem.h>
8  #include <sys/shm.h>
9  #include <sys/time.h>
10
11 #define DEFAULT_FILENAME "readfile.txt"
12 #define BUFFER_SIZE 100
13 #define ITERATIONS_COUNT 10
14
15 static struct sembuf WAIT[1] = {1, -1, 0};
16 static struct sembuf RELEASE[1] = {2, 1, 0};
17 static struct sembuf LOCK[1] = {0, -1, 0};
18 static struct sembuf UNLOCK[1] = {0, 1, 0};
19
20 int* buffer;
21 int shmemory;
22 int semaphore;
23
24 // Обработчик сигнала прерывания необходим( для корректного завершения работы при прерывании)
25 void signalHandler(int sig);
26 // Функция для корректного завершения работы приложения
27 void clearAndExit(int code);
28
29 int main(int argc, char** argv) {
30   char* filename = DEFAULT_FILENAME;
31
32   if(argc < 2)
33     printf("Using default filename: %s.\n", filename);
34   else
```

```
35      strcpy(filename, argv[1]);
36
37    // Пробуем создать ключ
38    key_t key = ftok(filename, 'Q');
39    if(key == -1) {
40      perror("It's impossible to get key for key file.\n");
41      return 0x1;
42    }
43
44    // Создаем shm
45    shmemory = shmget(key, (BUFFER_SIZE + 1) * sizeof(int), 0666);
46    if(shmemory < 0) {
47      perror("It's impossible to create shm.\n");
48      return 0x2;
49    }
50
51    // Присоединяем shm в наше адресное пространство
52    buffer = (int*) shmat(shmemory, 0, 0);
53    if(buffer < 0) {
54      perror("It's impossible to attach shm.\n");
55      return 0x3;
56    }
57
58    // Обработка сигнала прерывания с терминала для корректного завершения работы
59    signal(SIGINT, signalHandler);
60
61    // Создание группы из двух семафоров: первый показывает, что можно читать, второй — что можно
        писать
62    semaphore = semget(key, 2, 0666);
63    if(semaphore < 0) {
64      perror("It's impossible to create semaphore.\n");
65      clearAndExit(0x4);
66    }
67
68    printf("Press \"Enter\" to start.\n");
69    getchar();
70
71    int send = 0;
72    for(int index = 0; index < ITERATIONS_COUNT; ++index) {
73      // Ожидаем хотя бы одну свободную ячейку
74      int result = semop(semaphore, WAIT, 1);
75      if(result < 0) {
76        perror("It's imposibble to wait not empty.\n");
77        clearAndExit(0x5);
78      }
79
80      // Ожидаем доступа к разделяемой памяти
81      result = semop(semaphore, LOCK, 1);
82      if(result < 0) {
83        perror("It's imposibble to set lock.\n");
84        clearAndExit(0x6);
85      }
86
87      ++buffer[BUFFER_SIZE];
88      printf("Add to cell #%d, value %d.\n", buffer[BUFFER_SIZE], send);
89      buffer[buffer[BUFFER_SIZE]] = send++;
90
91      // Ожидаем доступ к памяти
92      result = semop(semaphore, UNLOCK, 1);
93      if(result < 0) {
94        perror("It's imposibble to set unlock.\n");
95        clearAndExit(0x7);
96      }
97
98      // Увеличиваем число занятых ячеек
99      result = semop(semaphore, RELEASE, 1);
```

```
100     if(result < 0) {
101       perror("It's imposibble to release.\n");
102       clearAndExit(0x8);
103     }
104   }
105
106   return 0x0;
107 }
108
109 void signalHandler(int sig) {
110   // Корректно завершаем работу приложения
111   clearAndExit(0x9);
112 }
113
114 void clearAndExit(int code) {
115   // Устанавливаем обработчик сигнала по умолчанию
116   signal(SIGINT, SIG_DFL);
117
118   // Отключаем разделяемую память
119   int result = shmdt(buffer);
120   if(result < 0)
121     perror("It's impossible to detach shm.\n");
122
123   printf("Clear finished.\n");
124   exit(code);
125 }
```

Результаты работы программ:

```
1  stakenschneider@stakenschneider:~/temp$ ./p7.1.3.cl
2  Using default filename: readfile.txt.
3  Press "Enter" to start.
4
5  Add to cell #0, value 0.
6  Add to cell #1, value 1.
7  Add to cell #2, value 2.
8  Add to cell #3, value 3.
9  Add to cell #4, value 4.
10 Add to cell #5, value 5.
11 Add to cell #6, value 6.
12 Add to cell #7, value 7.
13 Add to cell #8, value 8.
14 Add to cell #9, value 9.
```

```
1  stakenschneider@stakenschneider:~/temp$ ./p7.1.3.cl
2  Using default filename: readfile.txt.
3  Press "Enter" to start.
4
5  Add to cell #10, value 0.
6  Add to cell #11, value 1.
7  Add to cell #12, value 2.
8  Add to cell #13, value 3.
9  Add to cell #14, value 4.
10 Add to cell #15, value 5.
11 Add to cell #16, value 6.
12 Add to cell #17, value 7.
13 Add to cell #18, value 8.
14 Add to cell #19, value 9.
```

```
1  stakenschneider@stakenschneider:~/temp$ ./p7.1.3.cl
2  Using default filename: readfile.txt.
3  Press "Enter" to start.
4
5  Add to cell #20, value 0.
6  Add to cell #21, value 1.
7  Add to cell #22, value 2.
```

```
Add to cell #23, value 3.
Add to cell #24, value 4.
Add to cell #25, value 5.
Add to cell #26, value 6.
Add to cell #27, value 7.
Add to cell #28, value 8.
Add to cell #29, value 9.
```

```
stakenschneider@stakenschneider:~/temp$ ./p7.1.3.se
Using default filename: readfile.txt.
Press "Enter" to start.

Remove from cell #29, value 9.
Remove from cell #28, value 8.
Remove from cell #27, value 7.
Remove from cell #26, value 6.
Remove from cell #25, value 5.
Remove from cell #24, value 4.
Remove from cell #23, value 3.
Remove from cell #22, value 2.
Remove from cell #21, value 1.
Remove from cell #20, value 0.
Remove from cell #19, value 9.
Remove from cell #18, value 8.
Remove from cell #17, value 7.
Remove from cell #16, value 6.
Remove from cell #15, value 5.
Remove from cell #14, value 4.
Remove from cell #13, value 3.
Remove from cell #12, value 2.
Remove from cell #11, value 1.
Remove from cell #10, value 0.
Remove from cell #9, value 9.
Remove from cell #8, value 8.
Remove from cell #7, value 7.
Remove from cell #6, value 6.
Remove from cell #5, value 5.
```

Программы клиенты записывают в буфер по десять значений. Каждый следующий клиент записывает в конец одного и того же буфера. Сервер считывает 25 чисел из тех 30, которые были добавлены тремя клиентами.

1.3.8 Глава 8. Сокеты

Сокеты - название программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут исполняться как на одной ЭВМ, так и на различных ЭВМ, связанных между собой сетью.

Следует различать клиентские и серверные сокеты. Клиентские сокеты можно сравнить с конечными аппаратами телефонной сети, а серверные — с коммутаторами. Клиентское приложение (например, браузер) использует только клиентские сокеты, а серверное (например, веб-сервер, которому браузер посылает запросы) — как клиентские, так и серверные сокеты.

Для создания сервера необходимо указать порт, который не должен быть занят. Для подключения клиента к серверу необходимо знать *IP* адрес и порт.

Для получения информации о состоянии сети используют утилиты *ifconfig* и *netstat*.

Утилиту *ifconfig* обычно используют для получения информации о внутренних и внешних *IP* адресах:

```
stakenschneider@stakenschneider:~/temp$ ifconfig
enp1s0    Link encap:Ethernet  HWaddr e8:9a:8f:e1:0e:17
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
```

```
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:4271 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4271 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:314292 (314.2 KB)  TX bytes:314292 (314.2 KB)

wlp2s0    Link encap:Ethernet  HWaddr 74:de:2b:64:22:23
          inet addr:192.168.0.106  Bcast:192.168.0.255  Mask:255.255.255.0
          inet6 addr: fe80::47b3:2637:b9d2:ef23/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3348 errors:0 dropped:0 overruns:0 frame:0
          TX packets:124 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1074445 (1.0 MB)  TX bytes:19056 (19.0 KB)
```

Утилиту *netstat* обычно используют для получения информации о количестве подключений, количестве отправленных и пришедших пакетов:

```
stakenschneider@stakenschneider:~/temp$ netstat -s
Ip:
    6777 total packets received
    2 with invalid addresses
    0 forwarded
    0 incoming packets discarded
    4260 incoming packets delivered
    4264 requests sent out
    1008 outgoing packets dropped
    2 dropped because of missing route
Icmp:
    2024 ICMP messages received
    0 input ICMP message failed.
    ICMP input histogram:
        destination unreachable: 2024
    2024 ICMP messages sent
    0 ICMP messages failed
    ICMP output histogram:
        destination unreachable: 2024
IcmpMsg:
        InType3: 2024
        OutType3: 2024
Tcp:
    14 active connections openings
    12 passive connection openings
    2 failed connection attempts
    0 connection resets received
    2 connections established
    137 segments received
    137 segments send out
    0 segments retransmited
    0 bad segments received.
    2 resets sent
Udp:
    76 packets received
    2024 packets to unknown port received.
    0 packet receive errors
    2100 packets sent
UdpLite:
TcpExt:
    11 TCP sockets finished time wait in fast timer
    26 packets directly queued to recvmsg prequeue.
    18 bytes directly received in process context from prequeue
    11 packet headers predicted
    33 acknowledgments not containing data payload received
    17 predicted acknowledgments
    TCPOrigDataSent: 59
```

```
IpExt:
    InMcastPkts: 103
    OutMcastPkts: 44
    InBcastPkts: 6
    InOctets: 1095501
    OutOctets: 316253
    InMcastOctets: 7152
    OutMcastOctets: 5267
    InBcastOctets: 468
    InNoECTPkts: 6777
```

1. Реализовать TCP сервер, который прослушивает заданный порт. При приходе нового соединения создается новый поток для его обработки.

Разработаем программу *TCP* сервера, который в бесконечном цикле ожидает подключения клиентов, создает для каждого из них новый поток, принимает сообщения клиента и отправляет их назад. Также реализована обработка сигнала прерывания для корректного завершения работы всех потоков и закрытия сокетов:

```
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <pthread.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <map>

#define PORT 65100

#define BACKLOG 5
#define BUFFER_SIZE 1000
#define FLAGS 0

// Коллекция для хранения пар значений:
// сокет + идентификатор потока
std::map<int, pthread_t> threads;
// Серверный сокет
int serverSocket;

// Обработчик сигнала прерывания корректное( завершение приложения)
void signalHandler(int sig);
// Обработчик клиентского потока
void* clientExecutor(void* clientSocket);
// Функция считывания строки символов с клиента
int readLine(int socket, char* buffer, int bufferSize, int flags);
// Функция отправки строки символов клиенту
int sendLine(int socket, char* buffer, int flags);
// Корректное закрытие сокета
void closeSocket(int socket);
// Завершение работы клиентского потока
void destroyClient(int socket);

int main(int argc, char** argv) {
  int port = PORT;
  if(argc < 2)
    printf("Using default port: %d.\n", port);
  else
    port = atoi(argv[1]);

  // Создание серверного сокета
  serverSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
  if(serverSocket < 0) {
```

```
47      perror("It's impossible to create socket");
48      return 0x1;
49    }
50
51    printf("Server socket %d created.\n", serverSocket);
52
53    // Структура, задающая адресные характеристики
54    struct sockaddr_in info;
55    info.sin_family = AF_INET;
56    info.sin_port = htons(port);
57    info.sin_addr.s_addr = htonl(INADDR_ANY);
58
59    // Биндим сервер на определенный адрес
60    int serverBind  = bind(serverSocket, (struct sockaddr *) &info, sizeof(info));
61    if(serverBind < 0) {
62      perror("It's impossible to bind socket");
63      return 0x2;
64    }
65
66    // Слушаем сокет
67    int serverListen = listen(serverSocket, BACKLOG);
68    if(serverListen != 0) {
69      perror("It's impossible to listen socket");
70      return 0x3;
71    }
72
73    // Обработка прерывания для корректного завершения приложения
74    signal(SIGINT, signalHandler);
75
76    printf("Wait clients.\n");
77
78    while(1) {
79      // Ждем подключения клиентов
80      int clientSocket = accept(serverSocket, NULL, NULL);
81
82      // Пробуем создать поток обработки клиентских сообщений
83      pthread_t thread;
84      int result = pthread_create(&thread, NULL, clientExecutor, (void *) &clientSocket);
85      if(result) {
86        perror("It's impossible to create new thread");
87        closeSocket(clientSocket);
88      }
89
90      // Добавляем в коллекцию пару значений: сокет + идентификатор потока
91      threads.insert(std::pair<int, pthread_t>(clientSocket, thread));
92    }
93
94    return 0x0;
95  }
96
97  void signalHandler(int sig) {
98    // Для всех элементов коллекции
99    for(std::map<int, pthread_t>::iterator current = threads.begin(); current != threads.
        end(); ++current) {
100     printf("Try to finish client with socket %d\n", current->first);
101     // Закрываем клиентские сокеты
102     closeSocket(current->first);
103     printf("Client socket %d closed.\n", current->first);
104   }
105
106   // Закрываем серверный сокет
107   closeSocket(serverSocket);
108   printf("Server socket %d closed.\n", serverSocket);
109
110   exit(0x0);
111 }
```

```
112
113 void* clientExecutor(void* socket) {
114   int clientSocket = *((int*) socket);
115
116   printf("Client thread with socket %d created.\n", clientSocket);
117
118   char buffer[BUFFER_SIZE];
119   while(1) {
120     // Ожидаем прибытия строки
121     int result = readLine(clientSocket, buffer, BUFFER_SIZE, FLAGS);
122     if(result < 0)
123       destroyClient(clientSocket);
124
125     if(strlen(buffer) <= 1)
126       destroyClient(clientSocket);
127
128     printf("Client message: %s\n", buffer);
129
130     // Отправляем строку назад
131     result = sendLine(clientSocket, buffer, FLAGS);
132     if(result < 0)
133       destroyClient(clientSocket);
134   }
135 }
136
137 int readLine(int socket, char* buffer, int bufferSize, int flags) {
138   // Очищаем буфер
139   bzero(buffer, bufferSize);
140
141   char resolvedSymbol = ' ';
142   for(int index = 0; index < BUFFER_SIZE; ++index) {
143     // Считываем по одному символу
144     int readSize = recv(socket, &resolvedSymbol, 1, flags);
145     if(readSize <= 0)
146         return -1;
147     else if(resolvedSymbol == '\n')
148         break;
149     else if(resolvedSymbol != '\r')
150         buffer[index] = resolvedSymbol;
151   }
152
153   return 0x0;
154 }
155
156 int sendLine(int socket, char* buffer, int flags) {
157   unsigned int length = strlen(buffer);
158
159   // Перед отправкой сообщения добавляем в конец перевод строки
160   if(length == 0)
161     return -1;
162   else if(buffer[length - 1] != '\n') {
163     if(length >= BUFFER_SIZE)
164       return -1;
165     else
166       buffer[length] = '\n';
167
168   }
169
170   length = strlen(buffer);
171
172   // Отправляем строку клиенту
173   int result = send(socket, buffer, length, flags);
174   return result;
175 }
176
177 void closeSocket(int socket) {
```

```
178   // Завершение работы сокета
179   int socketShutdown = shutdown(socket, SHUT_RDWR);
180       if(socketShutdown != 0)
181     perror("It's impossible to shutdown socket");
182
183   // Закрытие сокета
184   int socketClose = close(socket);
185       if(socketClose != 0)
186     perror("It's impossible to close socket");
187 }
188
189 void destroyClient(int socket) {
190   printf("It's impossible to receive message from client or send message to client.\n");
191   // Завершение работы сокета
192   closeSocket(socket);
193   // Удаление пары значений из коллекции по ключю
194   threads.erase(socket);
195   printf("Client socket %d closed.\n", socket);
196   // Завершение работы потока
197   pthread_exit(NULL);
198 }
```

Клиент подключается к серверу, считывает сообщения из консоли и отправляет их серверу, после этого ожидает ответного сообщения сервера. Если было введено пустое сообщение, то клиент завершает работу:

```
1  #include <arpa/inet.h>
2  #include <netinet/in.h>
3  #include <stdio.h>
4  #include <string.h>
5  #include <stdlib.h>
6  #include <unistd.h>
7  #include <sys/types.h>
8  #include <signal.h>
9
10 #define PORT 65100
11 #define IP "127.0.0.1"
12
13 #define BACKLOG 5
14 #define BUFFER_SIZE 1000
15 #define IP_SIZE 16
16 #define FLAGS 0
17
18 // Клиентский сокет
19 int clientSocket;
20
21 // Обработчик сигнала прерывания корректное( завершение приложения)
22 void signalHandler(int sig);
23 // Функция считывания строки символов с сервера
24 int readLine(int socket, char* buffer, int bufferSize, int flags);
25 // Функция отправки строки символов серверу
26 int sendLine(int socket, char* buffer, int flags);
27 // Корректное закрытие сокета
28 void closeSocket(int socket);
29
30 int main(int argc, char** argv) {
31   int port = PORT;
32   char ip[IP_SIZE];
33
34   strcpy(ip, IP);
35   if(argc < 3) {
36     printf("Using default ip: %s.\n", ip);
37     printf("Using default port: %d.\n", port);
38   }
39   else {
40     strcpy(ip, argv[1]);
41     port = atoi(argv[2]);
42   }
```

```
43
44    // Создание клиентского сокета
45    clientSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
46    if(clientSocket < 0) {
47      perror("It's impossible to create socket");
48      return 0x1;
49    }
50
51    printf("Client socket %d created.\n", clientSocket);
52
53    // Структура, задающая адресные характеристики
54    struct sockaddr_in info;
55    info.sin_family = AF_INET;
56    info.sin_port = htons(port);
57    info.sin_addr.s_addr = inet_addr(ip);
58
59    // Подключение к серверу
60    int clientConnect = connect(clientSocket, (struct sockaddr *) &info, sizeof(info));
61    if(clientConnect) {
62      perror("It's impossible to connect");
63      return 0x2;
64    }
65
66    printf("Connection established.\n");
67
68    // Обработка прерывания для корректного завершения приложения
69    signal(SIGINT, signalHandler);
70
71    printf("Ready to send messages.\n");
72
73    char buffer[BUFFER_SIZE];
74    while(1) {
75      bzero(buffer, BUFFER_SIZE);
76      fgets(buffer, BUFFER_SIZE, stdin);
77
78      if(strlen(buffer) == 0 || buffer[0] == '\n') {
79        printf("Empty message, try to close client socket.\n");
80        closeSocket(clientSocket);
81        return 0x0;
82      }
83
84      // Отправляем строку на сервер
85      int result = sendLine(clientSocket, buffer, FLAGS);
86      if(result < 0) {
87        printf("It's impossible to send message, try to close client socket.\n");
88        closeSocket(clientSocket);
89        return 0x0;
90      }
91
92      // Ожидаем ответ строки
93      result = readLine(clientSocket, buffer, BUFFER_SIZE, FLAGS);
94      if(result < 0) {
95        printf("It's impossible to receive message, try to close client socket.\n");
96        closeSocket(clientSocket);
97        return 0x0;
98      }
99
100     printf("Server message: %s\n", buffer);
101   }
102
103   return 0x0;
104 }
105
106 void signalHandler(int sig) {
107   // Закрываем клиентский сокет
108   closeSocket(clientSocket);
```

```
109   printf("Client socket %d closed.\n", clientSocket);
110
111   exit(0x0);
112 }
113
114 int readLine(int socket, char* buffer, int bufferSize, int flags) {
115   // Очищаем буфер
116   bzero(buffer, bufferSize);
117
118   char resolvedSymbol = ' ';
119   for(int index = 0; index < BUFFER_SIZE; ++index) {
120     // Считываем по одному символу
121     int readSize = recv(socket, &resolvedSymbol, 1, flags);
122     if(readSize <= 0)
123         return -1;
124     else if(resolvedSymbol == '\n')
125         break;
126     else if(resolvedSymbol != '\r')
127         buffer[index] = resolvedSymbol;
128   }
129
130   return 0x0;
131 }
132
133 int sendLine(int socket, char* buffer, int flags) {
134   unsigned int length = strlen(buffer);
135
136   // Перед отправкой сообщения добавляем в конец перевод строки
137   if(length == 0)
138     return -1;
139   else if(buffer[length - 1] != '\n') {
140     if(length >= BUFFER_SIZE)
141       return -1;
142     else
143       buffer[length] = '\n';
144
145   }
146
147   length = strlen(buffer);
148
149   // Отправляем строку серверу
150   int result = send(socket, buffer, length, flags);
151   return result;
152 }
153
154 void closeSocket(int socket) {
155   // Завершение работы сокета
156   int socketShutdown = shutdown(socket, SHUT_RDWR);
157       if(socketShutdown != 0)
158     perror("It's impossible to shutdown socket");
159
160   // Закрытие сокета
161   int socketClose = close(socket);
162       if(socketClose != 0)
163     perror("It's impossible to close socket");
164 }
```

Протестируем клиент-серверное приложение:

```
1 stakenschneider@stakenschneider:~/temp$ g++ -pthread p8.1.se.cpp -o p8.1.se
2 stakenschneider@stakenschneider:~/temp$ ./p8.1.se
3 Using default port: 65100.
4 Server socket 3 created.
5 Wait clients.
6 Client thread with socket 4 created.
7 Client message: hello
8 Client message: how are you
```

```
Client message: bye bye, next message is empty
It's impossible to receive message from client or send message to client.
Client socket 4 closed.
```

```
stakenschneider@stakenschneider:~/temp$ gcc p8.1.cl.c -o p8.1.cl
stakenschneider@stakenschneider:~/temp$ ./p8.1.cl
Using default ip: 127.0.0.1.
Using default port: 65100.
Client socket 3 created.
Connection established.
Ready to send messages.
hello
Server message: hello
how are you
Server message: how are you
bye bye, next message is empty
Server message: bye bye, next message is empty

Empty message, try to close client socket.
```

В первую очередь был запущен *TCP* сервер. После этого к нему подключился клиент, отправил набор сообщений, сообщения вернулись назад клиенту. После ввода пустой строки клиент завершает работу и сервер сразу же узнает об этом.

Теперь рассмотрим подключение нескольких клиентов:

```
stakenschneider@stakenschneider:~/temp$ ./p8.1.se
Using default port: 65100.
Server socket 3 created.
Wait clients.
Client thread with socket 4 created.
Client thread with socket 5 created.
Client message: hello1
Client message: hello2
Client message: im ready to disconnect on signal ctrl+c
Client message: yeah it's good idea
^CTry to finish client with socket 4
Client socket 4 closed.
Try to finish client with socket 5
Client socket 5 closed.
Server socket 3 closed.
stakenschneider@stakenschneider:~/temp$
```

```
# Terminal 1
stakenschneider@stakenschneider:~/temp$ ./p8.1.cl
Using default ip: 127.0.0.1.
Using default port: 65100.
Client socket 3 created.
Connection established.
Ready to send messages.
hello1
Server message: hello1
im ready to disconnect on signal ctrl+c
Server message: im ready to disconnect on signal ctrl+c
last message when server is down
It's impossible to receive message, try to close client socket.
It's impossible to shutdown socket: Transport endpoint is not connected
stakenschneider@stakenschneider:~/temp$


# Terminal 2
stakenschneider@stakenschneider:~/temp$ ./p8.1.cl
Using default ip: 127.0.0.1.
Using default port: 65100.
Client socket 3 created.
Connection established.
Ready to send messages.
```

```
hello2
Server message: hello2
yeah it's good idea
Server message: yeah it's good idea
last message when server is down
It's impossible to receive message, try to close client socket.
It's impossible to shutdown socket: Transport endpoint is not connected
stakenschneider@stakenschneider:~/temp$
```

Два клиента подключаются к *TCP* серверу и отправляют ему сообщения. После этого сервер корректно отключается обрабатываемым сигналом прерывания. Клиенты мгновенно узнают о потере соединения, так как сервер выполнил *shutdown* на все клиентские сокеты.

2. Модифицировать программу для работы с большим количеством клиентов. Провести эксперимент, определяющий при каком максимальнов количестве клиентов TCP сервер завершает работу.

Модифицируем программу-клиент таким образом, чтобы сообщения посылались не с консольного ввода, а из определенной константной строки. Сообщения присылаются один раз в несколько секунд и есть лимит на количество сообщений, после которого программа завершается:

```
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>

#define PORT 65100
#define IP "127.0.0.1"

#define BACKLOG 5
#define BUFFER_SIZE 1000
#define IP_SIZE 16
#define FLAGS 0

#define DELAY 5
#define ITERATIONS_COUNT 30

// Клиентский сокет
int clientSocket;

// Обработчик сигнала прерывания корректное( завершение приложения)
void signalHandler(int sig);
// Функция считывания строки символов с сервера
int readLine(int socket, char* buffer, int bufferSize, int flags);
// Функция отправки строки символов серверу
int sendLine(int socket, char* buffer, int flags);
// Корректное закрытие сокета
void closeSocket(int socket);

int main(int argc, char** argv) {
  int port = PORT;
  char ip[IP_SIZE];

  strcpy(ip, IP);
  if(argc < 3) {
    printf("Using default ip: %s.\n", ip);
    printf("Using default port: %d.\n", port);
  }
  else {
    strcpy(ip, argv[1]);
    port = atoi(argv[2]);
  }
```

```
46
47    // Создание клиентского сокета
48    clientSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
49    if(clientSocket < 0) {
50      perror("It's impossible to create socket");
51      return 0x1;
52    }
53
54    printf("Client socket %d created.\n", clientSocket);
55
56    // Структура, задающая адресные характеристики
57    struct sockaddr_in info;
58    info.sin_family = AF_INET;
59    info.sin_port = htons(port);
60    info.sin_addr.s_addr = inet_addr(ip);
61
62    // Подключение к серверу
63    int clientConnect = connect(clientSocket, (struct sockaddr *) &info, sizeof(info));
64    if(clientConnect) {
65      perror("It's impossible to connect");
66      return 0x2;
67    }
68
69    printf("Connection established.\n");
70
71    // Обработка прерывания для корректного завершения приложения
72    signal(SIGINT, signalHandler);
73
74    printf("Ready to send messages.\n");
75
76    char buffer[BUFFER_SIZE];
77    for(int index = 0; index < ITERATIONS_COUNT; ++index) {
78      bzero(buffer, BUFFER_SIZE);
79      strcpy(buffer, "Message to server.\n");
80
81      if(strlen(buffer) == 0 || buffer[0] == '\n') {
82        printf("Empty message, try to close client socket.\n");
83        closeSocket(clientSocket);
84        return 0x0;
85      }
86
87      // Отправляем строку на сервер
88      int result = sendLine(clientSocket, buffer, FLAGS);
89      if(result < 0) {
90        printf("It's impossible to send message, try to close client socket.\n");
91        closeSocket(clientSocket);
92        return 0x0;
93      }
94
95      // Ожидаем ответ строки
96      result = readLine(clientSocket, buffer, BUFFER_SIZE, FLAGS);
97      if(result < 0) {
98        printf("It's impossible to receive message, try to close client socket.\n");
99        closeSocket(clientSocket);
100       return 0x0;
101     }
102
103     printf("Server message: %s\n", buffer);
104
105     // Отправляем один раз в несколько секунд
106     sleep(DELAY);
107   }
108
109   return 0x0;
110 }
111
```

```
112 void signalHandler(int sig) {
113   // Закрываем клиентский сокет
114   closeSocket(clientSocket);
115   printf("Client socket %d closed.\n", clientSocket);
116 
117   exit(0x0);
118 }
119 
120 int readLine(int socket, char* buffer, int bufferSize, int flags) {
121   // Очищаем буфер
122   bzero(buffer, bufferSize);
123 
124   char resolvedSymbol = ' ';
125   for(int index = 0; index < BUFFER_SIZE; ++index) {
126     // Считываем по одному символу
127     int readSize = recv(socket, &resolvedSymbol, 1, flags);
128     if(readSize <= 0)
129         return -1;
130     else if(resolvedSymbol == '\n')
131         break;
132     else if(resolvedSymbol != '\r')
133         buffer[index] = resolvedSymbol;
134   }
135 
136   return 0x0;
137 }
138 
139 int sendLine(int socket, char* buffer, int flags) {
140   unsigned int length = strlen(buffer);
141 
142   // Перед отправкой сообщения добавляем в конец перевод строки
143   if(length == 0)
144     return -1;
145   else if(buffer[length - 1] != '\n') {
146     if(length >= BUFFER_SIZE)
147       return -1;
148     else
149       buffer[length] = '\n';
150 
151   }
152 
153   length = strlen(buffer);
154 
155   // Отправляем строку серверу
156   int result = send(socket, buffer, length, flags);
157   return result;
158 }
159 
160 void closeSocket(int socket) {
161   // Завершение работы сокета
162   int socketShutdown = shutdown(socket, SHUT_RDWR);
163       if(socketShutdown != 0)
164     perror("It's impossible to shutdown socket");
165 
166   // Закрытие сокета
167   int socketClose = close(socket);
168       if(socketClose != 0)
169     perror("It's impossible to close socket");
170 }
```

Был написан скрипт, который запускает заданное количество клиентов в фоновом режиме:

```
1 #!/bin/bash
2 
3 # Если пользователь не указал количество итераций, то выходим с ошибкой
4 
5 count=$1;
```

```
if [ -z $count ]; then
  exit 1
fi

# Компилируем программу клиент
gcc p8.2.cl.c -o p8.2.cl

# Запускаем программыклиенты-
for i in $(seq 1 $count)
do
  ./p8.2.cl &
done
```

Экспериментально было подтверждено, что сервер стабильно и правильно работает с количеством клиентов от 0 до 1024. На 1025 клиенте сервер начинает выдавать непредвиденные ошибки. Это связано с тем, что количество *TCP* клиентов на один сервер ограничено константой *__FD_SETSIZE*, определенной в файле *typesizes.h*:

```
stakenschneider@stakenschneider:/$ cat /usr/include/x86_64-linux-gnu/bits/typesizes.h |
    grep "FD_SETSIZE"
#define __FD_SETSIZE     1024
```

В данной системе эта константа по умолчанию имеет размер 1024, однако ее можно изменить, определив директивой *#define* перед библиотекой сокетов.

3. Выполнить аналогичные действия на основе протокола UDP, сравнить с очередями сообщений.

Разработаем программу *UDP* сервера. В отличии от *TCP*, соединение не устанавливается, поэтому отсутствуют функции *listen* и *accept*. Также не имеет большого смысла регистрировать отдельный поток для каждого клиента. Сервер в бесконечном цикле принимает сообщения от клиента и отправляет их назад:

```
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <pthread.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <map>

#define PORT 65100

#define BACKLOG 5
#define BUFFER_SIZE 1000
#define FLAGS 0

// Серверный сокет
int serverSocket;

// Обработчик сигнала прерывания корректное( завершение приложения)
void signalHandler(int sig);
// Функция отправки строки символов клиенту
int sendLine(char* buffer, int flags, const sockaddr_in* address);

int main(int argc, char** argv) {
  int port = PORT;
  if(argc < 2)
    printf("Using default port: %d.\n", port);
  else
    port = atoi(argv[1]);

  // Создание серверного сокета
  serverSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
  if(serverSocket < 0) {
```

```
36      perror("It's impossible to create socket");
37      return 0x1;
38    }
39
40    printf("Server socket %d created.\n", serverSocket);
41
42    // Структура, задающая адресные характеристики
43    struct sockaddr_in info;
44    info.sin_family = AF_INET;
45    info.sin_port = htons(port);
46    info.sin_addr.s_addr = htonl(INADDR_ANY);
47
48    // Биндим сервер на определенный адрес
49    int serverBind  = bind(serverSocket, (struct sockaddr *) &info, sizeof(info));
50    if(serverBind < 0) {
51      perror("It's impossible to bind socket");
52      return 0x2;
53    }
54
55    // Обработка прерывания для корректного завершения приложения
56    signal(SIGINT, signalHandler);
57
58    printf("Wait clients.\n");
59
60    char buffer[BUFFER_SIZE];
61    while(1) {
62      struct sockaddr_in* address = new sockaddr_in;
63      size_t size = sizeof(struct sockaddr_in);
64
65      // Ожидаем прибытия строки
66      bzero(buffer, BUFFER_SIZE);
67      int result = recvfrom(serverSocket, buffer, BUFFER_SIZE, FLAGS, (sockaddr *) address,
        (socklen_t *) &size);
68      if(result < 0) {
69        delete address;
70        printf("It's impossible to receive message from client.\n");
71        continue;
72      }
73
74      if(strlen(buffer) <= 1) {
75        delete address;
76        printf("Message is empty.\n");
77        continue;
78      }
79
80      printf("Client message: %s", buffer);
81
82      // Отправляем строку назад
83      result = sendLine(buffer, FLAGS, address);
84      if(result < 0) {
85        delete address;
86        printf("It's impossible to send message to client.\n");
87        continue;
88      }
89
90      delete address;
91    }
92
93    return 0x0;
94  }
95
96  void signalHandler(int sig) {
97    // Закрываем серверный сокет
98    int socketClose = close(serverSocket);
99        if(socketClose != 0)
100     perror("It's impossible to close socket");
```

```
101   else
102     printf("Server socket %d closed.\n", serverSocket);
103
104   exit(0x0);
105 }
106
107 int sendLine(char* buffer, int flags, const sockaddr_in* address) {
108   unsigned int length = strlen(buffer);
109
110   // Перед отправкой сообщения добавляем в конец перевод строки
111   if(length == 0)
112     return -1;
113   else if(buffer[length - 1] != '\n') {
114     if(length >= BUFFER_SIZE)
115       return -1;
116     else
117       buffer[length] = '\n';
118
119   }
120
121   length = strlen(buffer);
122
123   // Отправляем строку клиенту
124   int result = sendto(serverSocket, buffer, length, flags, (struct sockaddr *) address,
        sizeof(struct sockaddr_in));
125   return result;
126 }
```

Клиент подключается к серверу, считывает сообщения из консоли и отправляет их серверу, после этого ожидает ответного сообщения сервера. В отличие от *TCP* отсутствует функция *connect*, отвечающая за установление соединения. Если было введено пустое сообщение, то клиент завершает работу:

```
1  #include <arpa/inet.h>
2  #include <netinet/in.h>
3  #include <stdio.h>
4  #include <string.h>
5  #include <stdlib.h>
6  #include <unistd.h>
7  #include <sys/types.h>
8  #include <signal.h>
9
10 #define PORT 65100
11 #define IP "127.0.0.1"
12
13 #define BACKLOG 5
14 #define BUFFER_SIZE 1000
15 #define IP_SIZE 16
16 #define FLAGS 0
17
18 // Клиентский сокет
19 int clientSocket;
20
21 // Обработчик сигнала прерывания корректное( завершение приложения)
22 void signalHandler(int sig);
23 // Функция отправки строки символов серверу
24 int sendLine(char* buffer, int flags, const struct sockaddr_in* address);
25 // Корректное закрытие сокета
26 void closeSocket(int socket);
27
28 int main(int argc, char** argv) {
29   int port = PORT;
30   char ip[IP_SIZE];
31
32   strcpy(ip, IP);
33   if(argc < 3) {
34     printf("Using default ip: %s.\n", ip);
35     printf("Using default port: %d.\n", port);
```

```
36    }
37    else {
38      strcpy(ip, argv[1]);
39      port = atoi(argv[2]);
40    }
41
42    // Создание клиентского сокета
43    clientSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
44    if(clientSocket < 0) {
45      perror("It's impossible to create socket");
46      return 0x1;
47    }
48
49    printf("Client socket %d created.\n", clientSocket);
50
51    // Структура, задающая адресные характеристики
52    struct sockaddr_in address;
53    address.sin_family = AF_INET;
54    address.sin_port = htons(port);
55    address.sin_addr.s_addr = inet_addr(ip);
56
57    printf("Connection established.\n");
58
59    // Обработка прерывания для корректного завершения приложения
60    signal(SIGINT, signalHandler);
61
62    printf("Ready to send messages.\n");
63
64    char buffer[BUFFER_SIZE];
65
66    while(1) {
67      bzero(buffer, BUFFER_SIZE);
68      fgets(buffer, BUFFER_SIZE, stdin);
69
70      if(strlen(buffer) == 0 || buffer[0] == '\n') {
71        printf("Empty message, try to close client socket.\n");
72        closeSocket(clientSocket);
73        return 0x0;
74      }
75
76      // Отправляем строку на сервер
77      int result = sendLine(buffer, FLAGS, &address);
78      if(result < 0) {
79        printf("It's impossible to send message, try to close client socket.\n");
80        closeSocket(clientSocket);
81        return 0x0;
82      }
83
84      size_t size = sizeof(struct sockaddr_in);
85
86      // Ожидаем ответ сервера
87      bzero(buffer, BUFFER_SIZE);
88      result = recvfrom(clientSocket, buffer, BUFFER_SIZE, FLAGS, (struct sockaddr *) &
        address, (socklen_t *) &size);
89      if(result < 0) {
90        printf("It's impossible to receive message, try to close client socket.\n");
91        closeSocket(clientSocket);
92        return 0x0;
93      }
94
95      printf("Server message: %s", buffer);
96    }
97
98    return 0x0;
99  }
100
```

```
101 void signalHandler(int sig) {
102   // Закрываем клиентский сокет
103   closeSocket(clientSocket);
104   printf("Client socket %d closed.\n", clientSocket);
105 
106   exit(0x0);
107 }
108 
109 int sendLine(char* buffer, int flags, const struct sockaddr_in* address) {
110   unsigned int length = strlen(buffer);
111 
112   // Перед отправкой сообщения добавляем в конец перевод строки
113   if(length == 0)
114     return -1;
115   else if(buffer[length - 1] != '\n') {
116     if(length >= BUFFER_SIZE)
117       return -1;
118     else
119       buffer[length] = '\n';
120 
121   }
122 
123   length = strlen(buffer);
124 
125   // Отправляем строку серверу
126   int result = sendto(clientSocket, buffer, length, flags, (struct sockaddr *) address,
      sizeof(struct sockaddr_in));
127   return result;
128 }
129 
130 void closeSocket(int socket) {
131   // Закрытие сокета
132   int socketClose = close(socket);
133       if(socketClose != 0)
134     perror("It's impossible to close socket");
135 }
```

Протестируем клиент-серверное приложение:

```
1  stakenschneider@stakenschneider:~/temp$ ./p8.3.se
2  Using default port: 65100.
3  Server socket 3 created.
4  Wait clients.
5  Client message: hello2
6  Client message: hello1
7  Client message: some text there!
8  Client message: oups!
9  ^CServer socket 3 closed.
10 stakenschneider@stakenschneider:~/temp$
```

```
1  # Terminal 1
2  stakenschneider@stakenschneider:~/temp$ gcc p8.3.cl.c -o p8.3.cl
3  stakenschneider@stakenschneider:~/temp$ ./p8.3.cl
4  Using default ip: 127.0.0.1.
5  Using default port: 65100.
6  Client socket 3 created.
7  Connection established.
8  Ready to send messages.
9  hello1
10 Server message: hello1
11 some text there!
12 Server message: some text there!
13 
14 # Terminal 2
15 stakenschneider@stakenschneider:~/temp$ ./p8.3.cl
16 Using default ip: 127.0.0.1.
```

```
Using default port: 65100.
Client socket 3 created.
Connection established.
Ready to send messages.
hello2
Server message: hello2
oups!
Server message: oups!
```

В первую очередь был запущен сервер два клиента. Клиенты отправляют сообщения серверу, после чего сервер завершает свою работу. В связи с тем что отсутствует установление соединения *UDP*, клиенты не узнают о том, что сервер отключился. Для обхода такой ситуации обычно с какой то очередностью серверу отправляется пакет, если от сервера пакет не пришел - значит сервер недоступен.

4. Провести эксперимент, определяющий при каком максимальнов количестве клиентов UDP сервер завершает работу.

Модифицируем программу-клиент таким образом, чтобы сообщения посылались не с консольного ввода, а из определенной константной строки. Сообщения присылаются один раз в несколько секунд и есть лимит на количество сообщений, после которого программа завершается:

```c
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>

#define PORT 65100
#define IP "127.0.0.1"

#define BACKLOG 5
#define BUFFER_SIZE 1000
#define IP_SIZE 16
#define FLAGS 0

#define DELAY 4
#define ITERATIONS_COUNT 15

// Клиентский сокет
int clientSocket;

// Обработчик сигнала прерывания корректное( завершение приложения)
void signalHandler(int sig);
// Функция отправки строки символов серверу
int sendLine(char* buffer, int flags, const struct sockaddr_in* address);
// Корректное закрытие сокета
void closeSocket(int socket);

int main(int argc, char** argv) {
  int port = PORT;
  char ip[IP_SIZE];

  strcpy(ip, IP);
  if(argc < 3) {
    printf("Using default ip: %s.\n", ip);
    printf("Using default port: %d.\n", port);
  }
  else {
    strcpy(ip, argv[1]);
    port = atoi(argv[2]);
  }

  // Создание клиентского сокета
```

```
46    clientSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
47    if(clientSocket < 0) {
48      perror("It's impossible to create socket");
49      return 0x1;
50    }
51
52    printf("Client socket %d created.\n", clientSocket);
53
54    // Структура, задающая адресные характеристики
55    struct sockaddr_in address;
56    address.sin_family = AF_INET;
57    address.sin_port = htons(port);
58    address.sin_addr.s_addr = inet_addr(ip);
59
60    printf("Connection established.\n");
61
62    // Обработка прерывания для корректного завершения приложения
63    signal(SIGINT, signalHandler);
64
65    printf("Ready to send messages.\n");
66
67    char buffer[BUFFER_SIZE];
68    for(int index = 0; index < ITERATIONS_COUNT; ++index) {
69      bzero(buffer, BUFFER_SIZE);
70      strcpy(buffer, "Message to server.\n");
71
72      if(strlen(buffer) == 0 || buffer[0] == '\n') {
73        printf("Empty message, try to close client socket.\n");
74        closeSocket(clientSocket);
75        return 0x0;
76      }
77
78      // Отправляем строку на сервер
79      int result = sendLine(buffer, FLAGS, &address);
80      if(result < 0) {
81        printf("It's impossible to send message, try to close client socket.\n");
82        closeSocket(clientSocket);
83        return 0x0;
84      }
85
86      size_t size = sizeof(struct sockaddr_in);
87
88      // Ожидаем ответ сервера
89      bzero(buffer, BUFFER_SIZE);
90      result = recvfrom(clientSocket, buffer, BUFFER_SIZE, FLAGS, (struct sockaddr *) &
        address, (socklen_t *) &size);
91      if(result < 0) {
92        printf("It's impossible to receive message, try to close client socket.\n");
93        closeSocket(clientSocket);
94        return 0x0;
95      }
96
97      printf("Server message: %s", buffer);
98
99      // Отправляем один раз в несколько секунд
100     sleep(DELAY);
101   }
102
103   closeSocket(clientSocket);
104
105   return 0x0;
106 }
107
108 void signalHandler(int sig) {
109   // Закрываем клиентский сокет
110   closeSocket(clientSocket);
```

```
111   printf("Client socket %d closed.\n", clientSocket);
112
113   exit(0x0);
114 }
115
116 int sendLine(char* buffer, int flags, const struct sockaddr_in* address) {
117   unsigned int length = strlen(buffer);
118
119   // Перед отправкой сообщения добавляем в конец перевод строки
120   if(length == 0)
121     return -1;
122   else if(buffer[length - 1] != '\n') {
123     if(length >= BUFFER_SIZE)
124       return -1;
125     else
126       buffer[length] = '\n';
127
128   }
129
130   length = strlen(buffer);
131
132   // Отправляем строку серверу
133   int result = sendto(clientSocket, buffer, length, flags, (struct sockaddr *) address,
        sizeof(struct sockaddr_in));
134   return result;
135 }
136
137 void closeSocket(int socket) {
138   // Закрытие сокета
139   int socketClose = close(socket);
140       if(socketClose != 0)
141     perror("It's impossible to close socket");
142 }
```

Был написан скрипт, который запускает заданное количество клиентов в фоновом режиме:

```
1  #!/bin/bash
2
3  # Если пользователь не указал количество итераций, то выходим с ошибкой
4
5  count=$1;
6  if [ -z $count ]; then
7    exit 1
8  fi
9
10 # Компилируем программу клиент
11 gcc p8.4.cl.c -o p8.4.cl
12
13 # Запускаем программыклиенты-
14 for i in $(seq 1 $count)
15 do
16   ./p8.4.cl &
17 done
```

Экспериментально было подтверждено, что сервер стабильно и правильно работает с количеством клиентов от 0 до 2300, на большем количестве производительности не хватило. С увеличением количества клиентов заметно начала снижаться производительность. Есть подозрения, что количество *UDP* клиентов ограничено только производительностью компьютера.

1.4 Вывод

В ходе работы, были рассмотрены основные способы межпроцессного взаимодействия в ОС Unix: надежные и ненадежные сигналы, именованные и неименованные каналы, очереди сообщений, семафоры и разделяемая память, сокеты.

Именованные и неименованные каналы позволяют организовать однонаправленную передачу по типу *FIFO*. Для дуплексной передачи между двумя процессами используют два разнонаправленных канала. Из-за того, что в передаче данных задействовано ядро ОС, взаимодействие удаленных процессов через сеть с помощью каналов невозможно.

Время доставки сообщения в очереди сравнимо с временем доставки сигнала, однако сообщение несет обычно больше информации, чем сигнал. С помощью очереди сообщений проще организовать асинхронный обмен данными между процессами, чем с помощью каналов.

Разделяемая память и семафоры обычно используются вместе. Семафоры позволяют синхронизировать доступ к разделяемому ресурсу. Самые часто используемые семафоры - битовые семафоры (мьютексы), которые активно используются для синхронизации потоков.

Сокеты - это универсальное средство *IPC*, которое является универсальным. Это означает, что его можно использовать также и для передачи данных между двумя разными компьютерами, подключенными к общей сети. Сокеты так или иначе реализованы на большинстве ОС, и большая часть сетевых приложений использует эту технологию.