Санкт-Петербургский Политехнический Университет Петра Великого

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

Операционные системы

Отчет по лабораторной работе №4

Системное программирование в ОС семейства Unix

Работу выполнил:

Иванов А.А.

Группа: 43501/1

Преподаватель:

Малышев И.А.

Санкт-Петербург

2018

# Цель работы

Изучить следующие средства межпроцессного взаимодействия (IPC) в ОС LINUX:

1. Надежные и ненадежные сигналы
2. Именованные и неименованные каналы
3. Очереди сообщений
4. Семафоры и разделяемая память
5. Сокеты

# Программа работы

1. «Ненадежные» сигналы
   1. Составьте программу sig\_father.c, позволяющую изменить диспозицию сигналов, а именно, установить: обработчик пользовательских сигналов SIGUSR1 и SIGUSR2, реакцию по умолчанию на сигнал SIGINT, игнорирование сигнала SIGCHLD.
   2. Самостоятельно повторить эксперимент для других сигналов и процессов, порождаемых в разных файлах, а также для потоков одного и разных процессов.
2. «Надежные» сигналы
   1. Создать программу, позволяющую продемонстрировать возможность отложенной обработки (временного блокирования) сигнала (например, SIGINT).
   2. Измените обработчик так, чтобы отправка сигнала SIGINT производилась из обработчика функцией kill(SIGINT, getpid()).
3. Сигналы POSIX реального времени
   1. Провести эксперимент, позволяющий определить возможность организации очереди для различных типов сигналов, обычных и реального времени (более двух сигналов. Для этого увеличить вложенность вызовов обработчиков)
   2. Экспериментально подтвердить, что обработка равно приоритетных сигналов реального времени происходит в порядке очереди.
   3. Опытным путем подтвердить наличие приоритетов сигналов реального времени.
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 из канала информацию. В процессе-потомке будет происходить считывание данных из файла и запись в канал.
5. Именованные каналы
   1. Создать клиент-серверное приложение, демонстрирующее дуплексную передачу информацию двумя однонаправленными именованными каналами между клиентом и сервером.
6. Очереди сообщений
   1. Создать клиент-серверное приложение, демонстрирующее передачу информации между процессами посредством очередей сообщений.
7. Семафоры и разделяемая память
   1. Выполнить три поставленных задачи с семафорами и разделяемой памятью.
8. Сокеты
   1. Реализовать 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 соединения 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, чтобы реализовать обмен сервера с множеством клиентов. Количество клиентов: 10, 100, 1000.
   3. Выполнить аналогичное действия на основе протокола UDP, сравнить с очередями сообщений.

# Ход работы

## I. «Ненадежные» сигналы

### 1. Составьте программу sig\_father.c, 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 сигналов.

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- отцу сигнал SIGUSR1 и извещает об удачной или неудачной отправке указанного сигнала. Остальные сигналы генерируются из командной строки.

Была написана программа sig\_father.c соответствующая заданию.

|  |
| --- |
| #include <stdio.h>  #include <signal.h>  #include <unistd.h>  #include <stdlib.h>  static void sigHandler(int sig) { // обработчик сигнала  printf("Catched signal %i\n",sig); // номер сигнала  printf("Parent = %d\n",getpid()); // pid  signal(sig,SIG\_DFL); // восстановление диспозиции  }  int main(void){  printf("Father started: PID = %i, PPID = %i\n",getpid(),getppid()); // инф. о родителе  signal(SIGUSR1, sigHandler);// установка дисп. сигнала для SIGUSR1  signal(SIGUSR2, sigHandler);// SIGUSR2  signal(SIGINT, SIG\_DFL); // реакция по умолчанию  signal(SIGCHLD, SIG\_IGN); // игнорирование  int forkRes = fork(); // пораждаем процесс  if (forkRes == 0) { // если процесс-потомок  printf("Son started: PID = %i, PPID = %i\n", getpid(), getppid());// инф. о процессе  if (kill(getppid(), SIGUSR1) != 0) { // сигнал родительскому процессу  printf("Error while sending SIGUSR1\n");  exit(1);  }  printf("Successfully sent SIGUSR1\n"); // сообщение о удаче  return 0;  }  while(1) // ожидание сигнала  pause();  return 0;  } |

Результат исполнения программы:

|  |
| --- |
| andrey@Inc~/lab4 $ gcc sig\_father.c -o sig\_father  andrey@Inc~/lab4 $ ./sig\_father  Father started: PID = 20207, PPID = 1950  Son started: PID = 20208, PPID = 20207  Successfully sent SIGUSR1  Catched signal 10  Parent = 20207  // Второе окно терминала, в котором были введены следующие команды:  andrey@Inc~ $ kill -SIGUSR2 20207  andrey@Inc~ $ kill -SIGCHLD 20207  andrey@Inc~ $ kill -SIGINT 20207  // Второе окно терминала было закрыто  Catched signal 12  Parent = 20207 |

Сперва дочерний процесс выслал сигнал SIGUSR1 родительскому процессу, который в последствии был обработан. Далее было открыто второе окно терминала, в котором были отправлены сигналы SIGUSR2, SIGCHLD, SIGINT. Сигнал SIGUSR2 был обработан, и об этом было выведено сообщение. Сигнал SIGCHLD был проигнорирован, как и было указано в программе. Сигнал SIGINT привел к завершению работы, что является реакцией по умолчанию.

Теперь отправим два сигнала SIGUSR2.

|  |
| --- |
| andrey@Inc~/lab4 $ ./sig\_father&  [1] 1892  Father started: PID = 1892, PPID = 1764  Son started: PID = 1893, PPID = 1892  Catched signal 10  Parent = 1892  Successfully sent SIGUSR1  andrey@Inc~/lab4 $ jobs  [1]+ Выполняется ./sig\_father &  andrey@Inc~/lab4 $ kill -SIGUSR2 1892  Catched signal 12  Parent = 1892  andrey@Inc~/lab4 $ kill -SIGUSR2 1892  andrey@Inc~/lab4 $ jobs  [1]+ Определяемый пользователем сигнал 2 ./sig\_father |

По результатам выполнения, первый сигнал был обработан написанным обработчиком, а второй обработался по умолчанию. Это произошло из-за того, что, после обработки первого сигнала произошло восстановления обработчика по умолчанию.

### 2. Самостоятельно повторить эксперимент для других сигналов и процессов, порождаемых в разных файлах, а также для потоков одного и разных процессов

В следующем примере в нити был создан обработчик сигнала, а затем выслан соответствующий сигнал. Программа ch1\_p2\_1.cpp:

|  |
| --- |
| #include <stdio.h>  #include <pthread.h>  #include <signal.h>  #include <unistd.h>  pthread\_t t1;  void handler(){ // обработчик сигнала  printf("Signal catched!\n");  pthread\_exit(0);  }  void\* thread1(void\* par){ // нить  signal(SIGUSR2, handler); // установка обработчика  sleep(1); // пауза  kill(getpid(),SIGUSR2); // посылка сигнала  while(true){ // бесконечный цикл  printf("Thread 1\n"); // для вывода в терминал  sleep(1);  }  }  int main(void){  pthread\_create(&t1,NULL, &thread1, NULL); // создание нити  while(true){ // бесконечный цикл  printf("Main thread\n");// для вывода в терминал  sleep(1);  }  } |

Результат выполнения:

|  |
| --- |
| andrey@Inc~/lab4 $ ./ch1\_p2\_1  Main thread  Main thread  Thread 1  Signal catched!  Thread 1  Thread 1  Thread 1  Thread 1  ^C |

Хоть сигнал и был вызван в нити, это не повлияло на работу нити, напротив свою работу прекратил основной поток. Для контроля работы нити необходимо использовать системный вызов pthread\_kill(), далее приведен пример с его использованием. Программа ch1\_p2\_2.cpp:

|  |
| --- |
| #include <stdio.h>  #include <pthread.h>  #include <signal.h>  #include <unistd.h>  pthread\_t t1;  void handler(){ // обработчик сигнала  printf("Signal catched!\n");  pthread\_exit(0);  }  void\* thread1(void\* par){ // нить  signal(SIGUSR2, handler); // установка обработчика  while(true){ // бесконечный цикл  printf("Thread 1\n"); // для вывода в терминал  sleep(1);  }  }  int main(void){  pthread\_create(&t1,NULL, &thread1, NULL); // создание нити  sleep(1); // пауза  pthread\_kill(t1,SIGUSR2); // посылка сигнала  while(true){ // бесконечный цикл  printf("Main thread\n");// для вывода в терминал  sleep(1);  }  } |

Результат выполнения:

|  |
| --- |
| andrey@Inc~/lab4 $ ./ch1\_p2\_2  Thread 1  Main thread  Signal catched!  Main thread  Main thread  Main thread  Main thread  ^C |

Как видно по результатам, нить указанная в pthread\_kill() завершила свою работу, а основной поток продолжил свою работу.

Далее приведены листинги программ для теста скорости взаимодействия двух разных процессов.

Программа ch1\_p2\_3.cpp:

|  |
| --- |
| #include <stdio.h>  #include <signal.h>  #include <unistd.h>  #include <stdlib.h>  void sig\_accept(){  system (" echo 'Signal SIGUSR1 accepted.' & date +\"%Y-%m-%d %H:%M:%S,%3N\" ");  exit(0);  }  int main(void){  printf("Receiver pid=%d\n",getpid());  signal(SIGUSR1,sig\_accept);  sleep(60);  } |

Программа ch1\_p2\_4.cpp:

|  |
| --- |
| #include <stdio.h>  #include <signal.h>  #include <unistd.h>  #include <stdlib.h>  int main(int argc, char \*argv[]){  system (" echo 'Signal SIGUSR1 sent.' & date +\"%Y-%m-%d %H:%M:%S,%3N\" ");  kill(atoi(argv[1]),SIGUSR1);  } |

Результат работы:

|  |
| --- |
| andrey@Inc~/lab4 $ ./ch1\_p2\_3&  [1] 16373  Receiver pid=16373  andrey@Inc~/lab4 $ ./ch1\_p2\_4 16373  Signal SIGUSR1 sent.  2017-12-05 12:47:53,627  Signal SIGUSR1 accepted.  andrey@Inc~/lab4 $ 2017-12-05 12:47:53,630 |

В фоновом режиме была запущена программа ch1\_p2\_3, далее была запущена программа ch1\_p2\_4, которая выслала сигнал SIGUSR по указанному pid. Как видно задержка между отсылкой и приемом составила 3 миллисекунды.

## II. «Надежные» сигналы

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

Программа ch2\_p1.cpp:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <signal.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <unistd.h>  void (\*mysig (int sig, void (\*handler) (int))) (int) {  struct sigaction act; // структура  act.sa\_handler = handler; // обработчик  sigemptyset(&act.sa\_mask); // инициализирует набор сигналов  sigaddset(&act.sa\_mask, SIGINT);// добавляет в набор SIGINT  act.sa\_flags = 0; // иниц. доп. флага нулем  if(sigaction(sig, &act, 0) < 0) // изменение действий процесса при получении сигнала  return (SIG\_ERR);  return (act.sa\_handler); // вернуть измененный обработчик  }  void usrHandler(int sig) { // обработчик  if (sig != SIGUSR1) {  printf("Bad signal catched:%i\n", sig); // пришел сигнал SIGUSR1  return;  }  printf("SIGUSR1 catched\n"); // обработчик перехватил сигнал  sleep(6); // засыпает на 6 секунд  }  int main() {  mysig(SIGUSR1, usrHandler);  while(1)  pause();  return 0;  } |

Выполнение ch2\_p1.cpp:

|  |
| --- |
| andrey@Inc~/lab4 $ ./ch2\_p1 &  [1] 4071  andrey@Inc~/lab4 $ kill -sigusr1 %1  SIGUSR1 catched  andrey@Inc~/lab4 $ kill -sigint %1  andrey@Inc~/lab4 $ jobs  [1]+ Прерывание ./ch2\_p1  andrey@Inc~/lab4 $ jobs |

Как видно из результатов выполнения, программа была запущена в фоновом режиме, далее были посланы сигналы SIGUSR1 и SIGINT, программа корректно обработала оба сигнала.

### 2 Измените обработчик так, чтобы отправка сигнала SIGINT производилась из обработчика функцией kill(SIGINT, getpid()).

Программа ch2\_p2.cpp:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <signal.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <unistd.h>  void (\*mysig (int sig, void (\*handler) (int))) (int) {  struct sigaction act; // структура  act.sa\_handler = handler; // обработчик  sigemptyset(&act.sa\_mask); // инициализирует набор сигналов  sigaddset(&act.sa\_mask, SIGINT);// добавляет в набор SIGINT  act.sa\_flags = 0; // иниц. доп. флага нулем  if(sigaction(sig, &act, 0) < 0) // изменение действий процесса при получении сигнала  return (SIG\_ERR);  return (act.sa\_handler); // вернуть измененный обработчик  }  void usrHandler(int sig) { // обработчик  if (sig != SIGUSR1) {  printf("Bad signal catched:%i\n", sig); // пришел сигнал SIGUSR1  return;  }  printf("SIGUSR1 catched\n"); // обработчик перехватил сигнал  kill(getpid(),SIGINT);  sleep(6); // засыпает на 6 секунд  }  int main() {  mysig(SIGUSR1, usrHandler);  while(1)  pause();  return 0;  } |

Выполнение ch2\_p2.cpp:

|  |
| --- |
| andrey@Inc~/lab4 $ g++ -o ch2\_p2 ch2\_p2.cpp  andrey@Inc~/lab4 $ ./ch2\_p2&  [1] 4274  andrey@Inc~/lab4 $ kill -SIGUSR1 %1  andrey@Inc~/lab4 $ SIGUSR1 catched  jobs  [1]+ Выполняется ./ch2\_p2 &  andrey@Inc~/lab4 $ jobs  [1]+ Выполняется ./ch2\_p2 &  andrey@Inc~/lab4 $ jobs  [1]+ Выполняется ./ch2\_p2 &  andrey@Inc~/lab4 $ jobs  [1]+ Выполняется ./ch2\_p2 &  andrey@Inc~/lab4 $ jobs  [1]+ Выполняется ./ch2\_p2 &  andrey@Inc~/lab4 $ jobs  [1]+ Прерывание ./ch2\_p2  andrey@Inc~/lab4 $ jobs |

При генерации сигнала SIGINT из обработчика другого сигнала обработка сгенерированного сигнала задерживается до конца выполнения текущего обработчика.

### 3. Сравните обработчики надежных и ненадежных сигналов, анализ представьте в отчете.

Ненадежные сигналы – это сигналы, обработчик которых срабатывает один раз, после чего его нужно снова перезапустить внутри самого обработчика. Если же в этот момент придет новый сигнал, то он будет потерян, т.к. обработчик еще не перезапустился и сигнал не обработался. Такие сигналы не надежны, т.к. перехват сигнала и перезапуск обработчика не являются атомарными операциями. В то время как в надёжных сигналах такой проблемы нет, и сигналы могут быть обработаны в любой момент времени. В предыдущих нескольких опытах это как раз было продемонстрировано.

## III. Сигналы POSIX реального времени

Сигналы в диапазоне от SIGRTMIN до SIGRTMAX являются сигналами POSIX реального времени:

* Такие сигналы помещаются в очередь.
* Доставляются несколько раз, если были отправлены несколько раз.
* Имеют систему приоритетов в соответствии с номером сигнала (меньший номер означает высший приоритет)

### 1. Провести эксперимент, позволяющий определить возможность организации очереди для различных типов сигналов, обычных и реального времени (более двух сигналов. Для этого увеличить вложенность вызовов обработчиков)

### 2. Экспериментально подтвердить, что обработка равно приоритетных сигналов реального времени происходит в порядке очереди.

### 3. Опытным путем подтвердить наличие приоритетов сигналов реального времени.

Для выполнения данных заданий была написана следующая программа ch3\_p1.cpp:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #include <sys/signal.h>  int rec\_sig[22]; // массив сигналов  unsigned int nfree\_elem = 0; // номер элемента  void sighandler(int signo) { // обработчик  rec\_sig[nfree\_elem++] = signo; // сохраняем номер принятого сигнала  return;  }  int main(int argc, char \* argv[]) {  sigset\_t mask; // сигналы, которые следует блокировать  struct sigaction act; // обеспечивает контроль над сигналами  memset(&act, 0, sizeof(act));  sigemptyset(&mask); // иниц. набора сигналов пустми значениями    //добавляем в маску сигналы  sigaddset(&mask, SIGRTMIN);  sigaddset(&mask, SIGRTMIN + 1);  sigaddset(&mask, SIGRTMIN + 2);  sigaddset(&mask, SIGRTMIN + 3);  sigaddset(&mask, SIGUSR1);  sigaddset(&mask, SIGUSR1);  sigaddset(&mask, SIGUSR2);  sigaddset(&mask, SIGCHLD);  //устанавливаем обработчик для сигнала  act.sa\_handler = sighandler;  act.sa\_mask = mask;  sigaction(SIGRTMIN, &act, NULL);  sigaction(SIGRTMIN + 1, &act, NULL);  sigaction(SIGRTMIN + 2, &act, NULL);  sigaction(SIGRTMIN + 3, &act, NULL);  sigaction(SIGUSR1, &act, NULL);  sigaction(SIGUSR2, &act, NULL);  sigaction(SIGCHLD, &act, NULL);  //блокируем обработчик сигналов  sigprocmask(SIG\_BLOCK, &mask, NULL);  //посылаем сигналы самому себе в случайном порядке  raise(SIGRTMIN + 3);  raise(SIGUSR2);  raise(SIGRTMIN + 2);  raise(SIGRTMIN + 1);  raise(SIGUSR1);  raise(SIGCHLD);  raise(SIGUSR1);  raise(SIGRTMIN);  raise(SIGRTMIN + 1);  raise(SIGRTMIN);  raise(SIGUSR1);  raise(SIGCHLD);  raise(SIGUSR2);  raise(SIGUSR1);  raise(SIGRTMIN + 1);  raise(SIGRTMIN + 2);  raise(SIGCHLD);  raise(SIGRTMIN);  raise(SIGRTMIN + 3);  raise(SIGRTMIN);  raise(SIGCHLD);  raise(SIGUSR1);  //разблокируем сигналы, что вызовет их немедленную отправку  sigprocmask(SIG\_UNBLOCK, &mask, NULL);  //выводим сигналы в порядке их получения  for(int i=0;i<nfree\_elem;++i){  if (rec\_sig[i] == SIGUSR1) {  printf("Signal SIGUSR1\n");  } else if (rec\_sig[i] == SIGCHLD) {  printf("Signal SIGCHLD\n");  } else if (rec\_sig[i] == SIGRTMIN) {  printf("Signal SIGRTMIN\n");  } else if (rec\_sig[i] == SIGRTMIN + 1) {  printf("Signal SIGRTMIN + 1\n");  } else if (rec\_sig[i] == SIGRTMIN + 2) {  printf("Signal SIGRTMIN + 2\n");  } else if (rec\_sig[i] == SIGRTMIN + 3) {  printf("Signal SIGRTMIN + 3\n");  } else if (rec\_sig[i] == SIGUSR2) {  printf("Signal SIGUSR2\n");  }  }  return 0;  } |

Выполнение ch3\_p1.cpp:

|  |
| --- |
| andrey@Inc~/lab4 $ g++ -o ch3\_p1 ch3\_p1.cpp  andrey@Inc~/lab4 $ ./ch3\_p1  Signal SIGUSR1  Signal SIGUSR2  Signal SIGCHLD  Signal SIGRTMIN  Signal SIGRTMIN  Signal SIGRTMIN  Signal SIGRTMIN  Signal SIGRTMIN + 1  Signal SIGRTMIN + 1  Signal SIGRTMIN + 1  Signal SIGRTMIN + 2  Signal SIGRTMIN + 2  Signal SIGRTMIN + 3  Signal SIGRTMIN + 3 |

Сигналы были посланы случайным образом, и были доставлены одновременно, но как видно из результатов, сформировалась некоторая очередь.

1. Пользовательские сигналы SIGUSR1 ( SIGUSR2 ) слились.
2. POSIX сигналы выстроены согласно номерам, т. к. сигналы с меньшими номерами доставляются раньше сигналов с большими номерами.
3. POSIX сигналы были приняты столько раз, сколько отправлены

## IV. Неименованные каналы

Программные (неименованные) каналы – однонаправленные, используются только для связи родственных процессов, в принципе могут использоваться и неродственными процессами, если предоставить им возможность передавать друг другу дескрипторы (т.к. имен они не имеют). Неименованный канал создается посредством системного вызова pipe(2), который возвращает 2 файловых дескриптора filedes[1] для записи в канал и filedes[0] для чтения из канала.

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

Следующая программа организована так, что процесс-родитель создает неименованный канал, потомка, закрывает канал на запись и записывает полученный из канала текст в текстовой файл. Процесс-потомок производит чтение из файла и запись в файл.

Программа ch4\_p1.cpp:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <string.h>  int main(void) {  char fileToRead[32]="temp1";  char fileToWrite[32]="temp2";  int filedes[2];  if (pipe(filedes) < 0) { // создаем неименованный канал  printf("Father : can't create pipe\n");  return -1;  }  printf("Father : pipe is created\n");  if (fork() == 0) { // создаем процесс  close(filedes[0]); // закрываем канал на чтение  FILE \* f = fopen(fileToRead, "r");  if (!f) { // открываем файл  printf("Son: can`t open file to read\n");  return -1;  }  char buf[100];  int res;  while (!feof(f)) { // читаем  res = fread(buf, sizeof(char), 100, f);  write(filedes[1], buf, res); // отправляем в канал  }  close(f); // закрываем файл  close(filedes[1]); // закрываем канал на запись  return 0;  } else {  close(filedes[1]); // закрываем канал на запись  FILE \* f = fopen(fileToWrite, "w");// открываем файл на запись  if (!f) { //  printf("Father: can`t open file to write\n");  return -1;  }  printf("Read from pipe:\n");  char buf[100];  int res;  while(1) {  bzero(buf, 100);  res = read(filedes[0], buf, 100);// читаем данные с канала  if (!res)  break;  printf("%s", buf);  fwrite(buf, sizeof(char), res, f); // пишем их в файл  }  fclose(f); // закрываем файл на чтение  close(filedes[0]); // закрываем канал на чтение  return 0;  }  } |

Содержимое файла temp1:

|  |
| --- |
| первая строчка  какой-то текст  еще строчка  последняя строчка |

Выполнение ch4\_p1.cpp:

|  |
| --- |
| andrey@Inc~/lab4 $ ./ch4\_p1  Father : pipe is created  Read from pipe:  первая строчка  какой-то текст  еще строчка  последняя строчка |

Был создан канал, по которому в одностороннем порядке были переданы данные от одного процесса другому. Содержимое файла temp1 успешно вывелось в консоль и переписалось в файл temp2.

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

## V. Именованные каналы

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

В программе сервера создается 2 именованных канала, используя системный вызов mknod(), аргументы которого: имя файла FIFO в файловой системе; флаги владения, прав доступа (установим открытые для всех права доступа на чтение и на запись S\_IFIFO | 0666). Открывается один канал на запись (chan1), другой - на чтение (chan2). Записывается имя файла в канал 1 (для записи) функцией write(), далее из канала 2 читаются данные в зарезервированный ранее буфер и выводится в консоль.

Программа ch5\_server.cpp:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <string.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #define DEF\_FILENAME "testFile.txt" // данные  int main(int argc, char\*\* argv) {  char fileName[30];  if(argc < 2) {  printf("Using default file name '%s'\n",DEF\_FILENAME);  strcpy(fileName,DEF\_FILENAME);  }  else  strcpy(fileName,argv[1]);  // создаем два канала  int res = mknod("chan1",S\_IFIFO | 0666,0);  if(res) {  printf("Can't create first channel\n");  exit(1);  }  res = mknod("chan2",S\_IFIFO | 0666,0);  if(res) {  printf("Can't create second channel\n");  exit(1);  }  // открываем первый канал на запись  int chan1 = open("chan1",O\_WRONLY);  if(chan1 == -1) {  printf("Can't open channel 1 for writing\n");  exit(0);  }  // открываем второй канал на чтение  int chan2 = open("chan2",O\_RDONLY);  if(chan2 == -1) {  printf("Can't open channel 2 for reading\n");  exit(0);  }    // пишем имя файла в первый канал  write(chan1,fileName,strlen(fileName));  char buf [100];  for(;;) { // читаем содержимое файла из второго канала  bzero(buf,100);  res = read(chan2,buf,100);  if(res <= 0)  break;  printf("Part of file: %s\n");  }  close(chan1); // закрываем каналы  close(chan2);  unlink("chan1"); // разрываем связь(удаляем)  unlink("chan2");  return 0;  } |

В клиентской части программы происходит чтение имени файла из канала 1, затем открытие канала 2 и запись в него данных, прочитанных из файла.

Программа ch5\_client.cpp:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <string.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  int main() {  // каналы сервер уже создал, открываем их  int chan1 = open("chan1", O\_RDONLY); // 1 канал на чтение  if(chan1 == -1) {  printf("Can't open channel1 for reading\n");  exit(0);  }  int chan2 = open("chan2", O\_WRONLY); // 2 на запись  if(chan2 == -1) {  printf("Can't open channel 2 for reading\n");  exit(0);  }  char fileName[100]; //пременная имени файла  bzero(fileName,100);  int res = read(chan1,fileName,100); // считываем из 1 канала имя файла  if(res <= 0) {  printf("Can't read fileName from channel1\n");  exit(0);  }  FILE \* f = fopen(fileName,"r"); // открываем файл на чтение  if(!f) {  printf("Can't open file %s\n",fileName);  exit(0);  }  char buf[100];  while(!feof(f)) { // читаем из файла и пишем во второй канал  res = fread(buf,sizeof(char),100,f);// читаем данные из файла  write(chan2,buf,res); // пишем их в канал  }  fclose(f); // закрываем файл  close(chan1); // закрываем каналы  close(chan2);  return 0;  } |

Содержимое файла testFile.txt:

|  |
| --- |
| some text  and again  some text |

Результаты выполнения:

|  |
| --- |
| andrey@Inc~/lab4 $ ./ch5\_server&  [1] 20947  andrey@Inc~/lab4 $ Using default file name 'testFile.txt'  ./ch5\_client  Part of file: some text  and again  some text  [1]+ Done ./ch5\_server |

Несмотря на то, что именованные каналы являются отдельным типом файлов и могут быть видимы разными процессами даже в распределенной файловой системе, использование FIFO для взаимодействия удаленных процессов и обмена информацией между ними невозможно. Так как и в этом случае для передачи данных задействовано ядро. Создаваемый файл служит для получения данных о расположении FIFO в адресном пространстве ядра и его состоянии. Это можно проверить, добавив паузу в сервер (в момент, когда сервер отправит имя файла, а клиент откроет файл и отправит данные обратно). В этот самый момент можно проверить размер и содержимое файлов:

|  |
| --- |
| andrey@Inc~/lab4 $ ls –l  итого 260  ...  prw-r--r-- 1 ana ana 0 дек 4 19:12 chan1  prw-r--r-- 1 ana ana 0 дек 4 19:12 chan2  ... |

Несмотря на записанные данные их размер 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 не возможным использование каналов в межсетевом общении.

## VI. Очереди сообщений

Очередь сообщений находится в адресном пространстве ядра и имеет ограниченный размер. В отличие от каналов, которые обладают теми же самыми свойствами, очереди сообщений сохраняют границы сообщений. Это значит, что ядро ОС гарантирует, что сообщение, поставленное в очередь, не смешается с предыдущим или следующим сообщением при чтении из очереди. Кроме того, с каждым сообщением связывается его тип. Процесс, читающий очередь сообщений, может отбирать только сообщения заданного типа или все сообщения кроме сообщений заданного типа.

Очередь сообщений можно рассматривать как связный список сообщений. Каждое сообщение представляет собой запись, очереди сообщений автоматически расставляют границы между записями, аналогично тому, как это делается в дейтаграммах UDP. Для записи сообщения в очередь не требуется наличия ожидающего его процесса в отличие от неименованных каналов и FIFO, в которые нельзя произвести запись, пока не появится считывающий данные процесс. Поэтому процесс может записать в очередь какие-то сообщения, после чего они могут быть получены другим процессом в любое время, даже если первый завершит свою работу. С завершением процесса-источника данные не исчезают (данные, остающиеся в именованном или неименованном канале, сбрасываются, после того как все процессы закроют его).

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

Аналогично предыдущему разделу программа включает 2 файла: серверный и клиентский. В общем случае одновременно могут работать несколько клиентов. Серверный файл содержит:

* подключение библиотек (см. листинг ниже)
* обработчик сигнала SIGINT (с восстановлением диспозиции и удалением очереди сообщений системным вызовом msgctl() для корректного завершения сервера при получении сигнала SIGINT);
* основную программу

далее создается очередь сообщений, используя системный вызов msgget(key, PERM | IPC\_CREAT), организовывается цикл ожидания сообщения и его чтение. Сервер в цикле читает сообщения из очереди (тип = 1) функцией msgrcv() и посылает на каждое сообщение ответ клиенту (тип = 2) функцией msgsnd() . Целесообразно дублировать вывод сообщений на экран для контроля. В случае возникновения любых ошибок функцией kill() инициируется посылка сигнала SIGINT. Обработчик сигнала выполняет восстановление диспозиции сигналов и удаление очереди сообщений системным вызовом msgctl(). В файле ch6\_client.cpp аналогично серверному коду должен быть получен ключ, затем доступ к очереди сообщений, отправка сообщения серверу (тип 1). Затем организовывается цикл ожидания сообщения клиентом с последующим чтением (тип 2). Таким образом, функции чтения и отправки сообщения реализуются системными вызовами: msgrcv(),msgsnd().

Описание работы сервера: сервер получает ключ по имени файла, с помощью ключа и идентификатора =’Q’ получает очередь сообщений и ждет сообщений с типом 1 от клиентов. При получении сообщения сервер выводит его в терминал и отсылает обратное сообщение с типом 2, содержащее фразу «Ок».

Описание работы клиента: клиент получает ту же очередь, что и сервер и ждет ввода пользователя. Считав ввод, идет посылка сообщения с типом 1, содержащая считанные данные и ожидает от сервера подтверждения о принятии.

Программа ch6\_server.cpp:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #include <sys/ipc.h>  #include <sys/msg.h>  #include <sys/types.h>  #include <signal.h>  #define DEF\_KEY\_FILE "key"  typedef struct { // структура для сообщения  long type; // тип сообщения  char buf[100]; // сообщение  } Message;  int queue; // переменная для очереди  void intHandler(int sig) { // обработчик сигнала  signal(sig,SIG\_DFL);  if(msgctl(queue,IPC\_RMID,0) < 0) {  printf("Can't delete queue\n");  exit(1);  }  }  int main(int argc, char\*\* argv) {  char keyFile[100];  bzero(keyFile,100); // обнуление  if(argc < 2) { // стандартный ключ используем  printf("Using default key file %s\n",DEF\_KEY\_FILE);  strcpy(keyFile,DEF\_KEY\_FILE);  }else // используем ключ указанный при запуске  strcpy(keyFile,argv[1]);  key\_t key;  key = ftok(keyFile,'Q');// создание ключа  if(key == -1) { // обработка ошибки создания  printf("Can't get key for key file %s and id 'Q'\n",keyFile);  exit(1);  }  queue = msgget(key,IPC\_CREAT | 0666); // возвращаем идентификатор очереди  if (queue < 0) { // нельзя создать очередь  printf("Can't create queue\n");  exit(4);  }  // до этого момента вызывали exit() а не kill, т.к. очередь  // еще не было создана  signal(SIGINT,intHandler);// уст. диспозицию сигнала  Message mes;  int res;  for(;;) {  bzero(mes.buf,100); // обнуляем буфер  // получаем первое сообщение с типом 1  res = msgrcv(queue,&mes,sizeof(Message),1L,0);  if(res < 0) {  printf("Error while recving msg\n");  kill(getpid(),SIGINT);  }  printf("Client's request: %s\n",mes.buf);  // шлем клиенту сообщение с типом 2, что все ок  mes.type = 2L;  bzero(mes.buf,100);  strcpy(mes.buf,"OK");  res = msgsnd(queue,(void\*)&mes,sizeof(Message),0);  if(res != 0) {  printf("error while sending msg\n");  kill(getpid(),SIGINT);  }  }  return 0;  } |

Программа ch6\_client.cpp:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <sys/ipc.h>  #include <sys/msg.h>  #include <sys/types.h>  #include <signal.h>  #define DEF\_KEY\_FILE "key"  typedef struct {  long type; // струтктура посылки  char buf[100];  } Message;  int queue;  int main(int argc, char\*\* argv) {  char keyFile[100];  bzero(keyFile,100);  if(argc < 2) { // выбираем файл  printf("Using default key file %s\n",DEF\_KEY\_FILE);  strcpy(keyFile,DEF\_KEY\_FILE);  }  else  strcpy(keyFile,argv[1]);  key\_t key;  key = ftok(keyFile,'Q');// создаем ключ  if(key == -1) {  printf("Can't get key for key file %s and id 'Q'\n",keyFile);  exit(1);  }  queue = msgget(key,0); // создаем очередь  if (queue < 0) {  printf("Can't create queue\n");  exit(4);  }  // основной цикл работы программы  Message mes;  int res;  for(;;) {  bzero(mes.buf,100);  // читаем сообщение с консоли  fgets(mes.buf,100,stdin);  mes.buf[strlen(mes.buf) - 1] = '\0';  // шлем его серверу  mes.type = 1L;  res = msgsnd(queue,(void\*)&mes,sizeof(Message),0);  if(res != 0) {  printf("Error while sending msg\n");  exit(1);  }  // получаем ответ, что все хорошо  res = msgrcv(queue,&mes,sizeof(Message),2L,0);  if(res < 0) {  printf("Error while recving msg\n");  exit(1);  }  printf("Server's response: %s\n",mes.buf);  }  return 0;  } |

Результаты выполнения ниже. Терминал с сервером:

|  |
| --- |
| andrey@Inc~/lab4 $ g++ -o ch6\_client ch6\_client.cpp  andrey@Inc~/lab4 $ g++ -o ch6\_server ch6\_server.cpp  andrey@Inc~/lab4 $ ./ch6\_server  Using default key file key  Client's request: hello world  Client's request: 1234567890  Client's request: some message  Client's request: another client  Client's request: hi bro  ^C  Error while recving msg |

Терминал с клиентом 1:

|  |
| --- |
| andrey@Inc~/lab4 $ ./ch6\_client  Using default key file key  hello world  Server's response: OK  1234567890  Server's response: OK  some message  Server's response: OK |

Терминал с клиентом 2:

|  |
| --- |
| andrey@Inc~/lab4 $ ./ch6\_client  Using default key file key  another client  Server's response: OK  hi bro  Server's response: OK |

Сообщения клиентов 1 и 2 успешно читались сервером, и они успешно получали сообщения с подтверждением принятия. По нажатию комбинации клавиш ctrl+c сервер завершил свою работу вывев в консоль сообщение "Error while recving msg". Если требуется определить некоторые константы, то это можно сделать через команду ipcs.

|  |
| --- |
| andrey@Inc~/lab4 $ ipcs -l  ------ Лимиты сообщений --------  максимум очередей для всей системы = 32000  max size of message (bytes) = 8192  default max size of queue (bytes) = 16384  ------ Пределы совм. исп. памяти --------  max number of segments = 4096  max seg size (kbytes) = 4177919  max total shared memory (kbytes) = 17112760316  min seg size (bytes) = 1  ------ Пределы семафоров --------  максимальное количество массивов = 32000  максимум семафоров на массив = 32000  максимум семафоров на всю систему = 1024000000  максимум операций на вызов семафора = 500  semaphore max value = 32767 |

Стоит отметить невозможность использования очереди сообщений для организации межпроцессного обмена через сеть, ввиду того, что очереди сообщений хранятся в адресном пространстве ядра, файл используется не как хранилище пересылаемых данных, а только для получения информации системой о них.

## VII. Семафоры и разделяемая память

Будут рассмотрены несколько вариантов поставленной задачи.

### Вариант 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.

Покажем, почему недостаточно одного на примере. Так как мы используем один семафор, то алгоритм работы читателя и писателя может быть только таким – захват семафора, выполнение действия (чтение / запись), освобождение семафора. Теперь допустим, что читатель прочитал данные, освободил семафор и еще не до конца использовал квант процессорного времени. Тогда он перейдет на новую итерацию, снова захватит только что освобожденный семафор и снова прочитает данные – ошибка.

Теперь покажем, почему достаточно двух семафоров. Придадим одному из них смысл «запись разрешена», т.е. читатель предыдущие данные уже использовал; второму – «чтение разрешено», т.е. писатель уже сгенерировал новые данные, которые нужно прочитать.

Оба семафора бинарные и используют стандартные операции, захват семафора – это ожидание освобождения ресурса (установки семафора в 1) и последующий захват ресурса (установки семафора в 0), освобождение ресурса – это установка семафора в 1.

Пару семафоров, использованных таким образом, 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.

Процесс-читатель ch7\_var1\_reader.cpp:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <signal.h>  #include <string.h>  #include <unistd.h>  #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/sem.h>  #include <sys/shm.h>  #include <sys/time.h>  #include "shm.h"  Message\* p\_msg;  int shmemory;  int semaphore;  void intHandler(int sig) {  //отключаем разделяемую память  if(shmdt(p\_msg) < 0) {  printf("Error while detaching shm\n");  exit(1);  }  //удаляем shm и семафоры  if(shmctl(shmemory, IPC\_RMID, 0) < 0) {  printf("Error while deleting shm\n");  exit(1);  }  if(semctl(semaphore, 0, IPC\_RMID) < 0) {  printf("Error while deleting semaphore\n");  exit(1);  }  }  int main(int argc, char\*\* argv) {  char keyFile[100];  bzero(keyFile,100);  if(argc < 2) {  printf("Using default key file %s\n",DEF\_KEY\_FILE);  strcpy(keyFile,DEF\_KEY\_FILE);  }  else  strcpy(keyFile,argv[1]);  key\_t key;  //будем использовать 1 и тот же ключ для семафора и для shm  if((key = ftok(keyFile, 'Q')) < 0) {  printf("Can't get key for key file %s and id 'Q'\n",keyFile);  exit(1);  }  //создаем shm  if((shmemory = shmget(key, sizeof(Message), IPC\_CREAT | 0666)) < 0) {  printf("Can't create shm\n");  exit(1);  }  //присоединяем shm в наше адресное пространство  if((p\_msg = (Message\*)shmat(shmemory, 0, 0)) < 0) {  printf("Error while attaching shm\n");  exit(1);  }  // устанавливаем обработчик сигнала  signal(SIGINT, intHandler);  //создаем группу из 2 семафоров  //1 - показывает, что можно читать  //2 - показывает, что можно писать  if((semaphore = semget(key, 2, IPC\_CREAT | 0666)) < 0) {  printf("Error while creating semaphore\n");  kill(getpid(),SIGINT);  }  // устнавливаем 2 семафор в 1, т.е. можно писать  if(semop(semaphore, setWriteEna, 1) < 0) {  printf("execution complete\n");  kill(getpid(),SIGINT);  }  // основной цикл работы  for(;;) {  // ждем пока клиент начнет работу  if(semop(semaphore, readEna, 1) < 0) {  printf("execution complete\n");  kill(getpid(),SIGINT);  }  //читаем сообщение от клиента  printf("Client's message: %s", p\_msg->buf);  // говорим клиенту, что можно снова писать  if(semop(semaphore, setWriteEna, 1) < 0) {  printf("execution complete\n");  kill(getpid(),SIGINT);  }  }  } |

Процесс-писатель ch7\_var1\_writer.cpp:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/sem.h>  #include <sys/shm.h>  #include <string.h>  #include "shm.h"  int main(int argc, char\*\* argv) {  Message\* p\_msg;  char keyFile[100];  bzero(keyFile,100);  if(argc < 2) {  printf("Using default key file %s\n",DEF\_KEY\_FILE);  strcpy(keyFile,DEF\_KEY\_FILE);  }  else  strcpy(keyFile,argv[1]);  key\_t key;  int shmemory;  int semaphore;  //будем использовать 1 и тот же ключ для семафора и для shm  if((key = ftok(keyFile, 'Q')) < 0) {  printf("Can't get key for key file %s and id 'Q'\n",keyFile);  exit(1);  }  //создаем shm  if((shmemory = shmget(key, sizeof(Message), 0666)) < 0) {  printf("Can't create shm\n");  exit(1);  }  //присоединяем shm в наше адресное пространство  if((p\_msg = (Message\*)shmat(shmemory, 0, 0)) < 0) {  printf("Error while attaching shm\n");  exit(1);  }  if((semaphore = semget(key, 2, 0666)) < 0) {  printf("Error while creating semaphore\n");  exit(1);  }  char buf[100];  for(;;) {  bzero(buf,100);  printf("Type message to serever. Empty string to finish\n");  fgets(buf,100,stdin);  if(strlen(buf) == 1 && buf[0] == '\n') {  printf("bye-bye\n");  exit(0);  }  //хотим отправить сообщение  if(semop(semaphore, writeEna, 1) < 0) {  printf("Can't execute a operation\n");  exit(1);  }  //запись сообщения в разделяемую память  sprintf(p\_msg->buf,"%s", buf);  //говорим серверу, что он может читать  if(semop(semaphore, setReadEna, 1) < 0) {  printf("Can't execute a operation\n");  exit(11);  }  }  //отключение от области разделяемой памяти  if(shmdt(p\_msg) < 0) {  printf("Error while detaching shm\n");  exit(1);  }  } |

Содержание shm.h:

|  |
| --- |
| #define DEF\_KEY\_FILE "key"  typedef struct {  long type;  char buf[100];  } Message;  static struct sembuf readEna[1] = {0,-1,0};  static struct sembuf writeEna[1] = {1,-1,0};  static struct sembuf setReadEna[1] = {0,1,0};  static struct sembuf setWriteEna[1] = {1,1,0}; |

Результаты работы.

Поток-писатель 1:

|  |
| --- |
| andrey@Inc~/lab4 $ ./ch7\_var1\_writer  Using default key file key  Type message to serever. Empty string to finish  12345  Type message to serever. Empty string to finish  first terminal  Type message to serever. Empty string to finish  bye-bye |

Поток-писатель 2:

|  |
| --- |
| andrey@Inc~/lab4 $ ./ch7\_var1\_writer  Using default key file key  Type message to serever. Empty string to finish  second terminal  Type message to serever. Empty string to finish  bye-bye |

Поток-читатель:

|  |
| --- |
| andrey@Inc~/lab4 $ ./ch7\_var1\_reader  Using default key file key  Client's message: 12345  Client's message: second terminal  Client's message: first terminal  ^Cexecution complete |

Все сообщения от клиента сервером прочитаны

### Вариант 2

К условиям предыдущей задачи добавляется условие корректной работы нескольких читателей и нескольких писателей одновременно. Как и в предыдущем варианте под чтением понимается извлечение данных из памяти, т. е. одну порцию данных может прочитать только один читатель.

Легко понять, что это условие не приводит к необходимости использования дополнительных средств синхронизации. Теперь вместо одного процесса, за каждый семафор будут конкурировать несколько. Но повторная запись или чтение все также невозможно. Так как, чтобы очередной процесс-писатель отработал, нужно освобождение семафора, которое выполняется из процесса-читателя, и наоборот.

### Вариант 3

К условиям предыдущей задачи добавляется наличие не единичного буфера, а буфера некоторого размера. Тип буфера (очередь, стек, кольцевой буфер) не имеет значения.

Двух семафоров по-прежнему достаточно, но это приведет к вырождению буфера, так как все процессы будут работать только с одной ячейкой.

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. Для этого выберем другие типы семафора и придадим им другую семантику. Возьмем два считающих семафора. Максимальное значение обоих – размер буфера. Первый инициализируется нулем и имеет смысл «количество заполненных ячеек», второй инициализируется N, где N – размер буфера и имеет смысл «количество пустых ячеек». Процессы-читатели перед своей работой захватывают семафор «количество заполненных ячеек», т.е. ждут появления хотя бы одной порции данных, а после чтения освобождают семафор «количество пустых ячеек». Процессы- писатели перед записью захватывают семафор «количество пустых ячеек», т.е. ждут появления хотя бы одной пустой ячейки для записи, а после записи освобождают семафор «количество полных ячеек». Таким образом, решается проблема чтения из пустого буфера и запись в полный.

Так как семафоры не бинарные, захватить их может сразу несколько процессов, т.е. несколько процессов попадут в секцию записи или чтения. В этом случае, если операция записи или чтения не атомарная (а зачастую так оно и есть), может произойти нарушение нормальной работы программы, к примеру, несколько процессов-писателей попытаются произвести запись в одну и ту же ячейку буфера, или несколько читателей выполнят чтение одной и той же ячейки. Таким образом, операции записи-чтения становятся критическими секциями, доступ к которым также необходимо синхронизировать. Для этого будет достаточно еще одного бинарного семафора, имеющего смысл «доступ к памяти разрешен». Оба типа процессов должны захватывать его при попытке взаимодействия с памятью и освобождать после.

Важно отметить, что порядок операций освобождения семафоров не важен, в то же время изменение порядка захвата семафоров может привести к взаимной блокировке процессов (dead lock). Взаимная блокировка в частности может произойти в таком случае: процесс- читатель захватил семафор «доступ к памяти разрешен», далее он проверяет, есть ли заполненные ячейки, т.е. пытается захватить семафор «количество заполненных ячеек», предположим, что свободных ячеек не оказалось и процесс переключился в неактивный режим, ожидая пока какой-нибудь процесс-писатель не запишет данные и не освободит семафор «количество заполненных ячеек». Но, этого никогда не произойдет, так как перед записью данных, процесс-писатель должен захватить семафор «доступ к памяти разрешен», который уже захвачен ожидающим процессом-читателем.

Пример решения последнего варианта задачи. В качестве разделяемого ресурса используется массив, находящийся в разделяемой памяти, ячейка памяти, находящаяся за последним элементом массива, интерпретируется как индекс последнего записанного элемента.

Исходный код программ:

Программа писатель ch7\_var3\_reader.cpp:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <signal.h>  #include <unistd.h>  #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/sem.h>  #include <sys/shm.h>  #include <sys/time.h>  #include <strings.h>  #include "shm\_2.h"  int\* buf;  int shmemory;  int semaphore;  void intHandler(int sig) {  shmdt(buf);  shmctl(shmemory, IPC\_RMID, 0);  semctl(semaphore, 0, IPC\_RMID);  }  int main(int argc, char\*\* argv) {  char keyFile[100];  bzero(keyFile,100);  if(argc < 2) {  printf("Using default key file %s\n",DEF\_KEY\_FILE);  strcpy(keyFile,DEF\_KEY\_FILE);  }  else  strcpy(keyFile,argv[1]);  key\_t key;  //будем использовать 1 и тот же ключ для семафора и для shm  if((key = ftok(keyFile, 'Q')) < 0) {  printf("Can't get key for key file %s and id 'Q'\n",keyFile);  exit(1);  }  //создаем shm  if((shmemory = shmget(key, (BUF\_SIZE+1)\*sizeof(int), IPC\_CREAT | 0666)) <0) {  printf("Can't create shm\n");  exit(1);  }  //присоединяем shm в наше адресное пространство  if((buf = (int\*)shmat(shmemory, 0, 0)) < 0) {  printf("Error while attaching shm\n");  exit(1);  }  // устанавливаем обработчик сигнала  signal(SIGINT, intHandler);  //создаем группу из 3 семафоров  //1 - число свободных ячеек  //2 - число занятых ячеек  // 3 работа с памятью  if((semaphore = semget(key, 3, IPC\_CREAT | 0666)) < 0) {  printf("Error while creating semaphore\n");  kill(getpid(),SIGINT);  }  // устанавливаем индекс в -1, первый клиент который напишет установит его в ноль  buf[BUF\_SIZE] = -1;  // инициализируем массив -1  int j = 0;  for(j = 0; j < BUF\_SIZE; ++j) {  buf[j] = -1;  }  // устнавливаем 1 семафор в число свободных ячеек, т.е. можно писать  if(semop(semaphore, setFree, 1) < 0) {  printf("execution complete\n");  kill(getpid(),SIGINT);  }  // говорим, что память свободна  if(semop(semaphore, mem\_unlock, 1) < 0) {  printf("execution complete\n");  kill(getpid(),SIGINT);  }  printf("Press enter to start working\n");  getchar();  // основной цикл работы  int i = 0;  for(i = 0; i < 15; ++i) {  // ждем пока будет хоть одна непустая ячейка  if(semop(semaphore, waitNotEmpty, 1) < 0) {  printf("execution complete\n");  kill(getpid(),SIGINT);  }  // ждем возможности поработать с памятью  if(semop(semaphore, mem\_lock, 1) < 0) {  printf("execution complete\n");  kill(getpid(),SIGINT);  }  //читаем сообщение от клиента  int res = buf[buf[BUF\_SIZE]];  buf[BUF\_SIZE] = buf[BUF\_SIZE] - 1;  printf("Remove %d from cell %d\n", res,buf[BUF\_SIZE]+1);  // освобождаем память  if(semop(semaphore, mem\_unlock, 1) < 0) {  printf("execution complete\n");  kill(getpid(),SIGINT);  }  // увеличиваем число пустых ячеек  if(semop(semaphore, releaseEmpty, 1) < 0) {  printf("execution complete\n");  kill(getpid(),SIGINT);  }  }  } |

Программа читатель ch7\_var3\_writer.cpp:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/sem.h>  #include <sys/shm.h>  #include <string.h>  #include "shm\_2.h"  int\* buf;  int main(int argc, char\*\* argv) {  char keyFile[100];  bzero(keyFile,100);  if(argc < 2) {  printf("Using default key file %s\n",DEF\_KEY\_FILE);  strcpy(keyFile,DEF\_KEY\_FILE);  }  else  strcpy(keyFile,argv[1]);  key\_t key;  int shmemory;  int semaphore;  //будем использовать 1 и тот же ключ для семафора и для shm  if((key = ftok(keyFile, 'Q')) < 0) {  printf("Can't get key for key file %s and id 'Q'\n",keyFile);  exit(1);  }  //создаем shm  if((shmemory = shmget(key, (BUF\_SIZE+1)\*sizeof(int), 0666)) < 0) {  printf("Can't create shm\n");  exit(1);  }  //присоединяем shm в наше адресное пространство  if((buf = (int\*)shmat(shmemory, 0, 0)) < 0) {  printf("Error while attaching shm\n");  exit(1);  }  if((semaphore = semget(key, 2, 0666)) < 0) {  printf("Error while creating semaphore\n");  exit(1);  }  printf("Press enter to start working\n");  getchar();  int send = 0;  char tb[10];  int i = 0;  for(i = 0; i < 10;++i) {  //ждем пока будет хоть одна свободная ячейка  if(semop(semaphore, waitNotFull, 1) < 0) {  printf("Can't execute a operation\n");  exit(1);  }  // ждем доступа к разделяемой памяти  if(semop(semaphore, mem\_lock, 1) < 0) {  printf("Can't execute a operation\n");  exit(1);  }  printf("Add %d to cell %d\n",send,buf[BUF\_SIZE]+1);  ++buf[BUF\_SIZE];  buf[buf[BUF\_SIZE]] = send++;  //освобождаем доступ к памяти  if(semop(semaphore, mem\_unlock, 1) < 0) {  printf("Can't execute a operation\n");  exit(11);  }  //увеличиваем число занятых ячеек  if(semop(semaphore, releaseFull, 1) < 0) {  printf("Can't execute a operation\n");  exit(11);  }  }  //отключение от области разделяемой памяти  shmdt(buf);  } |

Содержание shm\_2.h:

|  |
| --- |
| #define DEF\_KEY\_FILE "key2"  #define BUF\_SIZE 100  typedef struct {  long type;  char buf[BUF\_SIZE];  } Message;  static struct sembuf setFree[1] = {1, BUF\_SIZE, 0};  static struct sembuf waitNotEmpty[1] = {2, -1, 0};  static struct sembuf releaseEmpty[1] = {1, 1, 0};  static struct sembuf releaseFull[1] = {2, 1, 0};  static struct sembuf waitNotFull[1] = {1, -1, 0};  //блокируем память  static struct sembuf mem\_lock[1] = {0, -1, 0};  //говорим что память свободна  static struct sembuf mem\_unlock[1] = { 0, 1, 0}; |

Результаты работы. Поток-писатель 1:

|  |
| --- |
| andrey@Inc~/lab4 $ ./ch7\_var3\_writer  Using default key file key2  Press enter to start working  Add 0 to cell 0  Add 1 to cell 1  Add 2 to cell 2  Add 3 to cell 3  Add 4 to cell 4  Add 5 to cell 5  Add 6 to cell 6  Add 7 to cell 7  Add 8 to cell 8  Add 9 to cell 9 |

Поток-писатель 2:

|  |
| --- |
| andrey@Inc~/lab4 $ ./ch7\_var3\_writer  Using default key file key2  Press enter to start working  Add 0 to cell 10  Add 1 to cell 11  Add 2 to cell 12  Add 3 to cell 13  Add 4 to cell 14  Add 5 to cell 15  Add 6 to cell 16  Add 7 to cell 17  Add 8 to cell 18  Add 9 to cell 19 |

Поток-читатель:

|  |
| --- |
| andrey@Inc~/lab4 $ ./ch7\_var3\_reader  Using default key file key2  Press enter to start working  Remove 9 from cell 19  Remove 8 from cell 18  Remove 7 from cell 17  Remove 6 from cell 16  Remove 5 from cell 15  Remove 4 from cell 14  Remove 3 from cell 13  Remove 2 from cell 12  Remove 1 from cell 11  Remove 0 from cell 10  Remove 9 from cell 9  Remove 8 from cell 8  Remove 7 from cell 7  Remove 6 from cell 6  Remove 5 from cell 5 |

Программа работает верно: сначала в буфер записывается 10 чисел, а затем из буфера считывается 10 чисел. Поддерживается так же многопользовательский режим. В данном случае программы-писатели записали в массив числа, а затем программа читатель их оттуда считала.

## VIII. Сокеты

### 1. Реализовать 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 соединения 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.

Программа ch8\_p1\_server.cpp:

|  |
| --- |
| #include <sys/types.h>  #include <sys/socket.h>  #include <netinet/in.h>  #include <arpa/inet.h>  #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <string.h>  #include <unistd.h>  #define DEF\_PORT 8888 // номер порта  #define DEF\_IP "127.0.0.1" // ip адрес  int readFix(int sock, char\* buf, int bufSize, int flags) {  unsigned msgLength = 0;  int res = recv( sock, &msgLength, sizeof(unsigned), flags | MSG\_WAITALL );  if (res <= 0)  return res;  if(res > bufSize) {  printf("Recieved more data, then we can store, exiting\n");  exit(1);  }  return recv(sock, buf, msgLength, flags | MSG\_WAITALL);  }  int sendFix(int sock, char\* buf, int flags) {  unsigned msgLength = strlen(buf);  int res = send(sock, &msgLength, sizeof(unsigned), flags );  if (res <= 0)  return res;  send(sock, buf, msgLength, flags);  }  void\* clientHandler(void\* args) {  int sock = (int)args; // сокет  char buf[100]; // буффер данных  int res = 0;  for(;;) {  bzero(buf,100); // обнуление буфера  res = readFix(sock, buf,100, 0); // принять данные  if ( res <= 0) { // удача?  perror( "Can't recv data from client, ending thread\n" );  pthread\_exit(NULL);  }  printf( "Some client sent: %s\n",buf); // выводим данные  res = sendFix(sock,buf,0); // отправить ответ  if ( res <= 0 ) {  perror("send call failed" );  pthread\_exit(NULL);  }  }  }  int main( int argc, char\*\* argv) {  int port = 0;  if (argc < 2) { // проверка наличия аргументов  printf ("Using default port %d\n",DEF\_PORT);  port = DEF\_PORT;  }  else  port = atoi(argv[1]);  struct sockaddr\_in listenerinfo; // инициализация параметров  listenerinfo.sin\_family = AF\_INET; // режим  listenerinfo.sin\_port = htons( port ); // номер порта  listenerinfo.sin\_addr.s\_addr = htonl( INADDR\_ANY ); // ip  int listener = socket(AF\_INET, SOCK\_STREAM, 0 ); // прослушивающий сокет создаем  if ( listener < 0 ) { // проверка на ошибки  perror( "Can't create socket to listen: " );  exit(1);  }  // цепляем сокет к адресу  int res = bind(listener, ( struct sockaddr \*)&listenerinfo, sizeof( listenerinfo ) );  if ( res < 0 ) {  perror( "Can't bind socket" );  exit( 1 );  }  res = listen(listener,5); // переводим сокет в сост. прослушивания  if (res) {  perror("Error while listening:");  exit(1);  }  for(;;) {  int client = accept(listener, NULL, NULL ); // принимает клиентское подключение  pthread\_t thrd; // создаем отдельный поток под него  res = pthread\_create(&thrd, NULL, clientHandler, (void \*) (client));  if (res){  printf("Error while creating new thread\n");  }  }  return 0;  } |

Программа клиент ch8\_p1\_client.cpp:

|  |
| --- |
| #include <sys/types.h>  #include <sys/socket.h>  #include <netinet/in.h>  #include <pthread.h>  #include <arpa/inet.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #define DEF\_PORT 8888 // порт сервера  #define DEF\_IP "10.1.208.8" // адрес сервера  int readFix(int sock, char\* buf, int bufSize, int flags) {  unsigned msgLength = 0;  int res = recv( sock, &msgLength, sizeof(unsigned), flags | MSG\_WAITALL );  if (res <= 0)  return res;  if(res > bufSize) {  printf("Recieved more data, then we can store, exiting\n");  exit(1);  }  return recv(sock, buf, msgLength, flags | MSG\_WAITALL);  }  int sendFix(int sock, char\* buf, int flags) {  unsigned msgLength = strlen(buf);  int res = send(sock, &msgLength, sizeof(unsigned), flags );  if (res <= 0)  return res;  send(sock, buf, msgLength, flags);  }  int main( int argc, char\*\* argv) {  char\* addr;  int port;  char\* readbuf;  if(argc <3) { // проверка на наличие аргументов  printf("Using default port %d\n",DEF\_PORT);  port = DEF\_PORT;  }  else  port = atoi(argv[2]);  if(argc < 2) {  printf("Using default addr %s\n",DEF\_IP);  addr = DEF\_IP;  }  else  addr = argv[1];  struct sockaddr\_in peer; // инициализация параметров  peer.sin\_family = AF\_INET;  peer.sin\_port = htons( port );  peer.sin\_addr.s\_addr = inet\_addr( addr );  int sock = socket( AF\_INET, SOCK\_STREAM, 0 ); // создаем клиентский сокет  if ( sock < 0 ){  perror( "Can't create socket\n" );  exit( 1 );  } // конектимся  int res = connect( sock, ( struct sockaddr \* )&peer, sizeof( peer ) );  if (res){  perror( "Can't connect to server:" );  exit( 1 );  }  char buf[100];  for(;;) {  printf("Input request (empty to exit)\n");  bzero(buf,100);  fgets(buf, 100, stdin);  buf[strlen(buf)-1] = '\0';  if(strlen(buf) == 0) {  printf("Bye-bye\n");  return 0;  }  res = sendFix(sock, buf,0); // отправили сообщение  if ( res <= 0 ) {  perror( "Error while sending:" );  exit( 1 );  }  bzero(buf,100); // обнулили  res = readFix(sock, buf, 100, 0); // приняли сообщение  if ( res <= 0 ) {  perror( "Error while receiving:" );  exit(1);  }  printf("Server's response: %s\n",buf);  }  return 0;  } |

Результаты работы. Клиент 1:

|  |
| --- |
| andrey@Inc~/lab4 $ ./ch8\_p1\_client  Using default port 8888  Using default addr 127.0.0.1  Input request (empty to exit)  hello 1  Server's response: hello 1  Input request (empty to exit)  some text from client 1  Server's response: some text from client 1  Input request (empty to exit)  bye bye  Server's response: bye bye  Input request (empty to exit)  Bye-bye |

Клиент 2:

|  |
| --- |
| andrey@Inc~/lab4 $ ./ch8\_p1\_client  Using default port 8888  Using default addr 127.0.0.1  Input request (empty to exit)  hello 2  Server's response: hello 2  Input request (empty to exit)  hi client 1  Server's response: hi client 1  Input request (empty to exit)  bye  Server's response: bye  Input request (empty to exit)  Bye-bye |

Сервер:

|  |
| --- |
| andrey@Inc~/lab4 $ ./ch8\_p1\_server  Using default port 8888  Some client sent: hello 1  Some client sent: hello 2  Some client sent: some text from client 1  Some client sent: hi client 1  Some client sent: bye bye  Can't recv data from client, ending thread  : Success  Some client sent: bye  Can't recv data from client, ending thread  : Success  ^C |

Как видно из результатов работы, сервер получает сообщения, переданные клиентами, далее сервер отвечает тем-же самым сообщением, реализуя эхо-функциональность. При этом поддерживается работа с многими клиентами одновременно.

### 2. Модифицировать предыдущую программу, чтобы реализовать обмен сервера с множеством клиентов. Количество клиентов: 10, 100, 1000.

Для автоматизации код клиента был изменен таким образом, чтобы все время работы клиента, серверу отправлялось заранее написанное сообщение.

Программа ch8\_p2\_client.cpp:

|  |
| --- |
| #include <sys/types.h>  #include <sys/socket.h>  #include <netinet/in.h>  #include <arpa/inet.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #define DEF\_PORT 8888 // порт сервера  #define DEF\_IP "127.0.0.1" // адрес сервера  int readFix(int sock, char\* buf, int bufSize, int flags) {  unsigned msgLength = 0;  int res = recv( sock, &msgLength, sizeof(unsigned), flags | MSG\_WAITALL );  if (res <= 0)  return res;  if(res > bufSize) {  printf("Recieved more data, then we can store, exiting\n");  exit(1);  }  return recv(sock, buf, msgLength, flags | MSG\_WAITALL);  }  int sendFix(int sock, char\* buf, int flags) {  unsigned msgLength = strlen(buf);  int res = send(sock, &msgLength, sizeof(unsigned), flags );  if (res <= 0)  return res;  send(sock, buf, msgLength, flags);  }  int main( int argc, char\*\* argv) {  char\* addr;  int port;  char\* readbuf;  if(argc <3) { // проверка на наличие аргументов  printf("Using default port %d\n",DEF\_PORT);  port = DEF\_PORT;  }  else  port = atoi(argv[2]);  if(argc < 2) {  printf("Using default addr %s\n",DEF\_IP);  addr = DEF\_IP;  }  else  addr = argv[1];  struct sockaddr\_in peer; // инициализация параметров  peer.sin\_family = AF\_INET;  peer.sin\_port = htons( port );  peer.sin\_addr.s\_addr = inet\_addr( addr );  int sock = socket( AF\_INET, SOCK\_STREAM, 0 ); // создаем клиентский сокет  if ( sock < 0 ){  perror( "Can't create socket\n" );  exit( 1 );  } // конектимся  int res = connect( sock, ( struct sockaddr \* )&peer, sizeof( peer ) );  if (res){  perror( "Can't connect to server:" );  exit( 1 );  }  char buf[100] = {'s', 'o', 'm', 'e', ' ', 'm', 'e', 's', 's', 'a','g', 'e', };  for(;;) {  printf("Input request (empty to exit)\n");  if(strlen(buf) == 0) {  printf("Bye-bye\n");  return 0;  }  res = sendFix(sock, buf,0); // отправили сообщение  if ( res <= 0 ) {  perror( "Error while sending:" );  exit( 1 );  }  bzero(buf,12); // обнулили  res = readFix(sock, buf, 100, 0); // приняли сообщение  if ( res <= 0 ) {  perror( "Error while receiving:" );  exit(1);  }  printf("Server's response: %s\n",buf);  }  return 0;  } |

Также был написан скрипт clients\_generator.sh для автоматизации запуска приложений-клиентов.

|  |
| --- |
| #!/bin/sh  for i in $(seq 1 $1)  do  ./ch8\_p2\_client &  done |

Результаты работы:

* 10 - сервер исправно работает
* 100 - сервер исправно работает
* 1000 - сервер исправно работает
* >=1150 - при работе сервера возникают ошибки Error while creating new thread

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

Программа-сервера ch8\_p3\_server.cpp:

|  |
| --- |
| #include <iostream>  #include <sstream>  #include <string>  #include <time.h>  #include <arpa/inet.h>  #include <netinet/in.h>  #include <stdio.h>  #include <sys/types.h>  #include <sys/socket.h>  #include <unistd.h>  #include <stdlib.h>  #include <string.h>  #include <strings.h>  #include <string>  #include <iostream>  #include <stdbool.h>  #include <pthread.h>  #define SERV\_PORT 8888  #define SERV\_ADDR "127.0.0.1"  using namespace std;  bool Finish;  int ListenSocket;  class Message {  struct M {  int ttl;  int id;  };  public:  M Messages[1000];  Message() {  for (int i = 0; i < 1000; i++)  Messages[i].id = -1;  }  void Process() {  for (int i = 0; i < 1000; i++) {  if(Messages[i].id != -1) {  Messages[i].ttl--;  if (Messages[i].ttl < 0) {  cout<<"pocket id "<<Messages[i].id<<" lost"<<endl;  Messages[i].id = -1;  }  }  }  }  void Send(sockaddr\_in cl,string msg) {  int i;  for (i = 0; i < 1000; i++) {  if(Messages[i].id == -1)  break;  }  Messages[i].id = rand();  Messages[i].ttl = 10;  stringstream ss;  ss<<Messages[i].id;  string res = "#Send ", tmp;  ss>>tmp;  res+=tmp + " " + msg;  sendto(ListenSocket,res.c\_str(),strlen(res.c\_str()),0, (struct sockaddr \*)&cl,sizeof(cl));  }  string Receive(sockaddr\_in &cl) {  while (!Finish) {  char buff[10000] = "";  socklen\_t len = sizeof(cl);  int n = recvfrom(ListenSocket,buff,10000,0,(struct sockaddr \*)&cl,&len);  stringstream ss(buff),form;  string tmp;  ss>>tmp;  if(tmp == "#Send") {  int id;  ss>>id;  string tmp = "#Response ", res;  form<<id;  form>>res;  tmp += res;  sendto(ListenSocket,tmp.c\_str(),strlen(tmp.c\_str()),0,  (struct sockaddr \*)&cl,sizeof(cl));  res = string(buff);  while (res.at(0) != ' ')  res = res.erase(0,1);  res = res.erase(0,1);  while (res.at(0) != ' ')  res = res.erase(0,1);  res = res.erase(0,1);  return res;  }else if(tmp == "#Response"){  int i,id;  ss>>id;  for (i = 0; i < 1000; i++)  if(Messages[i].id == id) {  Messages[i].id = -1;  break;  }  }  }  return "";  }  };  Message mes;  int main(){  int n;  struct sockaddr\_in servaddr,cl;  socklen\_t len;  ListenSocket=socket(AF\_INET,SOCK\_DGRAM,0); // создаем сокет  bzero(&servaddr,sizeof(servaddr));  servaddr.sin\_family = AF\_INET; // инициализация  servaddr.sin\_addr.s\_addr=htonl(INADDR\_ANY);  servaddr.sin\_port=htons(SERV\_PORT);  bind(ListenSocket,(struct sockaddr \*)&servaddr,sizeof(servaddr)); // цепляемся к адресу  cout<<"Started"<<endl;  while (!Finish){  int len = sizeof(cl);  string rec = mes.Receive(cl); // принимаем сообщение  if (rec == "\n")  continue;  printf("Message from client: %s\n", rec.c\_str());  mes.Send(cl, "Какое-то сообщение клиенту"); // отправляем сообщение  printf("Message to client: Какое-то сообщение клиенту\n");  }  cout<<"Stoped"<<endl;  close(ListenSocket); // закрываем сокет  printf("Press any key\n");  std::cin.get();  return 0;  } |

Программа ch8\_p3\_client.cpp:

|  |
| --- |
| #include <arpa/inet.h>  #include <netinet/in.h>  #include <stdio.h>  #include <sys/types.h>  #include <sys/socket.h>  #include <unistd.h>  #include <stdlib.h>  #include <string.h>  #include <strings.h>  #include <string>  #include <iostream>  #include <stdbool.h>  #include <sstream>  #include <pthread.h>  #define MAX\_MSG 1000  #define BUF\_SIZE 10000 // размер буфера  #define SERV\_PORT 8888 // номер порта  #define SERV\_ADDR "10.1.208.8" // адрес  using namespace std;  bool Finish = false;  struct sockaddr\_in servaddr;  int sockfd;  class Message{ // класс передачи и приема  struct M {  int ttl;  int id;  string msg;  };  public:  M Messages[MAX\_MSG];  Message() { // конструктор(начальная иниц. буфера - свободен =-1)  for (int i = 0; i < MAX\_MSG; i++)  Messages[i].id = -1;  }  // ищем сообщения у которых закончился ttl - значит они потеряны(удаляем эти сообщения выводим инф о них на экран)  void Process() {  for (int i = 0; i < MAX\_MSG; i++) {  if(Messages[i].id != -1) {  Messages[i].ttl--;  if (Messages[i].ttl < 0) {  cout<<"Packet id="<<Messages[i].id<<" lost"<<endl;  Messages[i].id = -1;  }  }  }  }  // отправляем сообщение  void Send(string msg) {  int i;  for (i = 0; i < MAX\_MSG; i++)  if(Messages[i].id == -1)  break; // ищем номер свободный  Messages[i].id = rand(); // случайный id  Messages[i].ttl = 10; // время жизни 10 - стандартно  Messages[i].msg = msg; // текст сообщения  stringstream ss;  ss<<Messages[i].id;  string res = "#Send ",tmp; // заголовок пакета  ss>>tmp;  res+=tmp + " " + msg; // + id и текст сообщения  sendto(sockfd,res.c\_str(),strlen(res.c\_str()),0,(struct sockaddr \*)&servaddr,sizeof(servaddr)); // отправляем пакет  }  // прием  string Receive() {  while (!Finish) {  char buff[BUF\_SIZE] = "";  recvfrom(sockfd,buff,BUF\_SIZE,0,0,0); // приняли пакет  stringstream ss(buff),form;  string tmp;  ss>>tmp; // отделяем заголовок  if(tmp == "#Send") { // обычное сообщение  int id;  ss>>id;  string res = string(buff);  while (res.at(0) != ' ')//#Send удаляем заголовок  res = res.erase(0,1);  res = res.erase(0,1);  while (res.at(0) != ' ') // удаляем id  res = res.erase(0,1);  res = res.erase(0,1);  return res; // возвращаем принятое сообщение  } else if(tmp == "#Response") { // подтверждение нашего старого сообщения  int i,id;  ss>>id; // выделяем id  for (i = 0; i < MAX\_MSG; i++) { //ищем его в массиве сообщений  if(Messages[i].id == id) {  Messages[i].id = -1; // удаляем оттуда  break;  }  }  }  }  return "";  }  };  Message mes;  void SetServer(const char \* addr) {  if (inet\_addr(addr) != INADDR\_NONE)  servaddr.sin\_addr.s\_addr = inet\_addr(addr);  }  void \*LoopThread(void \*){ // задержка и вызов раз в секунду обработчика  while (!Finish) {  mes.Process();  usleep(100000);  }  }  int main() {  sockfd=socket(AF\_INET,SOCK\_DGRAM,0); // создаем сокет  bzero(&servaddr,sizeof(servaddr));  servaddr.sin\_family = AF\_INET; // инициализация  servaddr.sin\_port=htons(SERV\_PORT);  SetServer(string(SERV\_ADDR).c\_str());  pthread\_t loopThread; // создаем поток под обработчик ttl  pthread\_create(&loopThread, NULL, &LoopThread, NULL);  string \* msg = new string("Какое-то сообщение высланное серверу");  while (!Finish) {  mes.Send(\*msg); // отправка сообщения  string rec = mes.Receive(); // принятие сообщения  printf("Msg from server: %s\n", rec.c\_str()); // вывод сообщения  }  pthread\_join(loopThread, NULL); // блокируем поток  delete msg;  return 0;  } |

Общий класс передачи данных message.cpp:

|  |
| --- |
| class Message{ // класс передачи и приема  struct M {  int ttl;  int id;  string msg;  };  public:  M Messages[MAX\_MSG];  Message() { // конструктор(начальная иниц. буфера - свободен =-1)  for (int i = 0; i < MAX\_MSG; i++)  Messages[i].id = -1;  }  // ищем сообщения у которых закончился ttl - значит они потеряны(удаляем эти сообщения выводим инф о них на экран)  void Process() {  for (int i = 0; i < MAX\_MSG; i++) {  if(Messages[i].id != -1) {  Messages[i].ttl--;  if (Messages[i].ttl < 0) {  cout<<"Packet id="<<Messages[i].id<<" lost"<<endl;  Messages[i].id = -1;  }  }  }  }  // отправляем сообщение  void Send(string msg) {  int i;  for (i = 0; i < MAX\_MSG; i++)  if(Messages[i].id == -1)  break; // ищем номер свободный  Messages[i].id = rand(); // случайный id  Messages[i].ttl = 10; // время жизни 10 - стандартно  Messages[i].msg = msg; // текст сообщения  stringstream ss;  ss<<Messages[i].id;  string res = "#Send ",tmp; // заголовок пакета  ss>>tmp;  res+=tmp + " " + msg; // + id и текст сообщения  sendto(sockfd,res.c\_str(),strlen(res.c\_str()),0,(struct sockaddr \*)&servaddr,sizeof(servaddr)); // отправляем пакет  }  // прием  string Receive() {  while (!Finish) {  char buff[BUF\_SIZE] = "";  recvfrom(sockfd,buff,BUF\_SIZE,0,0,0); // приняли пакет  stringstream ss(buff),form;  string tmp;  ss>>tmp; // отделяем заголовок  if(tmp == "#Send") { // обычное сообщение  int id;  ss>>id;  string res = string(buff);  while (res.at(0) != ' ')//#Send удаляем заголовок  res = res.erase(0,1);  res = res.erase(0,1);  while (res.at(0) != ' ') // удаляем id  res = res.erase(0,1);  res = res.erase(0,1);  return res; // возвращаем принятое сообщение  } else if(tmp == "#Response") { // подтверждение нашего старого сообщения  int i,id;  ss>>id; // выделяем id  for (i = 0; i < MAX\_MSG; i++) { //ищем его в массиве сообщений  if(Messages[i].id == id) {  Messages[i].id = -1; // удаляем оттуда  break;  }  }  }  }  return "";  }  }; |

Как и раньше, работа сервера была протестирована на различном количестве клиентов, используя скрипт из предыдущего пункта. Терминал сервера:

|  |
| --- |
| andrey@Inc~/lab4 $ ./ch8\_p3\_server  Started  Message from client: Какое-то сообщение высланное серверу  Message to client: Какое-то сообщение клиенту  Message from client: Какое-то сообщение высланное серверу  Message to client: Какое-то сообщение клиенту  Message from client: Какое-то сообщение высланное серверу  Message to client: Какое-то сообщение клиенту  Message from client: Какое-то сообщение высланное серверу  ... |

Терминал клиента:

|  |
| --- |
| andrey@Inc~/lab4 $ ./udp\_clients\_generator.sh 1000  Msg from server: Какое-то сообщение клиенту  Msg from server: Какое-то сообщение клиенту  Msg from server: Какое-то сообщение клиенту  Msg from server: Какое-то сообщение клиенту  Msg from server: Какое-то сообщение клиенту  Msg from server: Какое-то сообщение клиенту  Msg from server: Какое-то сообщение клиенту  Msg from server: Какое-то сообщение клиенту  ... |

* 10 - сервер исправно работает
* 100 - сервер исправно работает
* 1000 - сервер исправно работает
* 2000 - сервер исправно работает
* >2000 - сервер исправно работает, но уже начинает чувствоваться некоторая заторможенность

Количество клиентов порядка двух тысяч не было проблемой для сервера, так как он реализован при помощи бесконечного цикла приема/обработки команд, а протокол UDP не требует создания подключения для каждого клиента. UDP структура более надежна в плане кол-ва клиентов, в связи с простотой реализации.

# Вывод

В данной лабораторной работе были рассмотрены основные механизмы межпроцессного взаимодействия в ОС Unix: сигналы, очереди сообщений, именованные и неименованные каналы, сокеты, разделяемая память и семафоры.

В процессе выполнения работы были рассмотрены основные механизмы межпроцессного взаимодействия в ОС Unix: сигналы, очереди сообщений, именованные и неименованные каналы, сокеты, разделяемая память и семафоры.

Именованные и неименованные каналы позволяют организовать однонаправленную передачу по типу FIFO. Поэтому для дуплексной передачи информации между двумя процессами необходимо два разнонаправленных канала. Также отличительной особенностью именованных каналов является то, что взаимодействие удаленных процессов через сеть невозможно посредством каналов, так как в этом случае для передачи данных задействовано ядро. Создаваемый файл служит для получения данных о расположении FIFO в адресном пространстве ядра и его состоянии.

Сообщения являются мощным средством межпроцессного обмена данными. Время доставки сообщения сравнимо с временем доставки сигнала, однако сообщение несёт гораздо больше информации, чем сигнал. С помощью сообщений гораздо проще организовать асинхронный обмен данными между процессами, чем с помощью каналов.

Семафоры и разделяемая память зачастую используются вместе. Семафоры позволяют синхронизировать доступ к разделяемому ресурсу и гарантировать «взаимное исключение» нескольких процессов при разделении ресурса (пока предыдущий процесс не закончит работу с ресурсом, следующий не начнет ее).

Сокеты являются средством IPC, которое можно использовать не только между процессами на одном компьютере, но и в сетевом режиме. Многие сетевые приложения построены на основе сокетов.

# 4. Список литературы

1. Душутина Е.В. «Межпроцессные взаимодействия в операционных система. Учебное пособие». СПб, 2014.