**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра МО ЭВМ**

**ОТЧЕТ**

**по лабораторной работе №5**

**по дисциплине «Операционные системы»**

**Тема: Средства межпроцессного взаимодействия в ОС семейства Unix.**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 1304 |  | Чернякова В.А. |
| Преподаватель |  | Душутина Е.В. |

Санкт-Петербург

2023

**Цель работы.**

Изучить системное программирование в ОС семейства UNIX.

**Выполнение работы.**

Модель ОС:

Linux Valeriya 4.15.0-142-generic #146~16.04.1-Ubuntu SMP Tue Apr 13 09:27:15 UTC 2021 x86\_64 x86\_64 x86\_64 GNU/Linux

1. **Сигналы.**

Сигналы позволяют осуществить самый примитивный способ коммуникации между двумя процессами. Сигналы в системе UNIX используются для того, чтобы: сообщить процессу о том, что возникло асинхронное событие; или необходимо обработать исключительное состояние.

Изначально сигналы были разработаны для уведомления об ошибках. В дальнейшем их стали использовать и как простейшую форму межпроцессного взаимодействия (IPС), например, для синхронизации процессов или для передачи простейших команд от одного процесса другому.

К генерации сигнала могут привести различные ситуации:

1. Ядро отправляет процессу (или группе процессов) сигнал при нажатии пользователем определенных клавиш или их комбинаций.

2. Аппаратные особые ситуации, например, деление на 0, обращение недопустимой области памяти и т.д., также вызывают генерацию сигнала. Обычно эти ситуации определяются аппаратурой компьютера, и ядру посылается соответствующее уведомление (например, виде прерывания). Ядро реагирует на это отправкой соответствующего сигнала процессу, который находился в стадии выполнения, когда произошла особая ситуация.

3. Определенные программные состояния системы или ее компонентов также могут вызвать отправку сигнала. В отличие от предыдущего случая, эти условия не связаны с аппаратной частью, а имеют программный характер. В качестве примера можно привести сигнал SIGALRM, отправляемый процессу, когда срабатывает таймер, ранее установленный с помощью вызова alarm().

Процесс может выбрать одно из трех возможных действий при получении сигнала:

1. игнорировать сигнал,

2. перехватить и самостоятельно обработать сигнал,

3. позволить действие по умолчанию.

Текущее действие при получении сигнала называется диспозицией сигнала.

Порожденный вызовом fork() процесс наследует диспозицию сигналов от своего родителя. Однако при вызове exec() диспозиция всех перехватываемых сигналов будет установлена ядром на действие по умолчанию.

* 1. **. Ненадёжные сигналы.**

Рассмотрим работу ненадежных сигналов.

Задание: Создадим программу, позволяющую изменить диспозицию сигналов. Установим обработчик пользовательских сигналов SIGUSR1 и SIGUSR2, реакцию по умолчанию на сигнал SIGINT и игнорирование сигнала SIGCHLD.

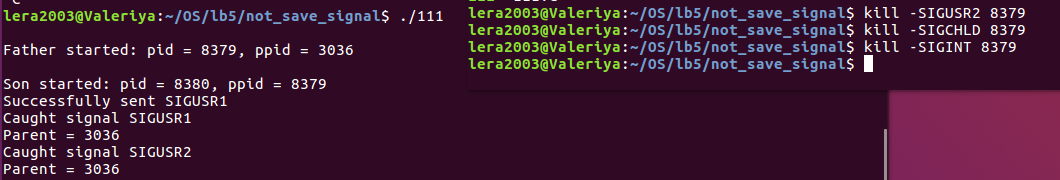
Породим процесс-копию и уйдём в ожидание сигналов.

Процесс-потомок, получив идентификатор родительского процесса, должен отправить процессу-отцу сигнал SIGUSR1 и извещение об удачной или неудачной отправке указанного сигнала.

*111.c*

|  |
| --- |
| #include <stdio.h>  #include <signal.h>  #include <unistd.h>  #include <stdlib.h>  #include <sys/wait.h>  static void sigHandler(int sig) {      printf("Caught signal %s\n", sig == SIGUSR1 ? "SIGUSR1" : "SIGUSR2");      printf("Parent = %d\n", getppid());      // Восстановление действия по умолчанию      signal(sig, SIG\_DFL);  }  int main() {      printf("\nFather started: pid = %i, ppid = %i\n", getpid(), getppid());      signal(SIGUSR1, sigHandler);      signal(SIGUSR2, sigHandler);      // SIGINT прерывание программы      // SIG\_DFL определяет заданное по умолчанию действие для специфического сигнала      signal(SIGINT, SIG\_DFL);      // SIGCHLD сигнал послан родительскому процессу всякий раз, когда один из дочерних процессов завершается или останавливается      // SIG\_IGN определяет, что сигнал должен игнорироваться      signal(SIGCHLD, SIG\_IGN);      int forkRes = fork();      if (forkRes == 0) {          // Child process          printf("\nSon started: pid = %i, ppid = %i\n", getpid(), getppid());          // Send signals to parent          if (kill(getppid(), SIGUSR1) != 0) {              printf("Error while sending SIGUSR1\n");              exit(1);          }          printf("Successfully sent SIGUSR1\n");          return 0;      }      // Parent process      wait(NULL); // Wait for child to finish      for (;;) {          pause(); // Wait for signals      }      return 0;  } |

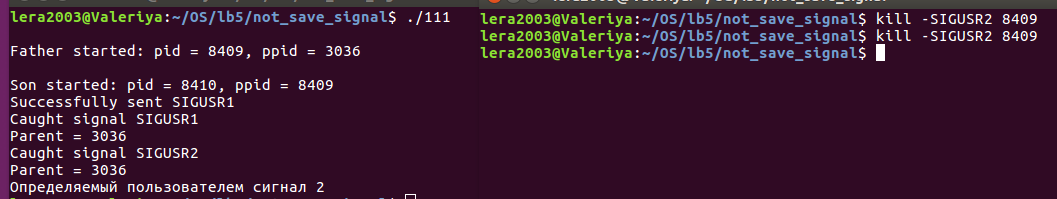
Результат работы программы:

****

Выводы: процесс-потомок отправил сигнал SIGUSR1, а процесс-отец его успешно принял. Сигнал SIGUSR2 также был «пойман», на сигнал SIGCHLD не последовало никакой реакции (так как он был проигнорирован), и сигнал SIGINT привел к завершению работы.

Запустим программу еще раз и дважды отправим ей сигнал SIGUSR2.

Результат работы программы:



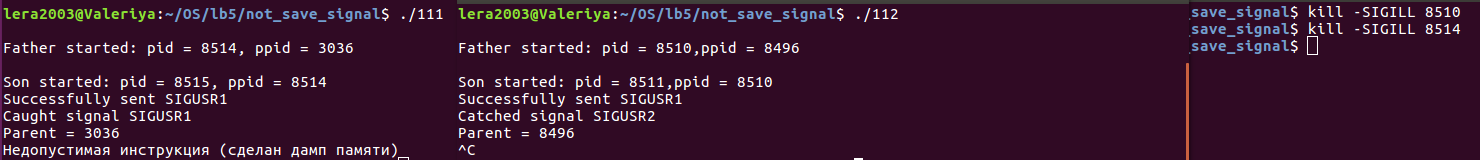
Выводы: в результате первый сигнал был «пойман», второй обработался по умолчанию. Это происходит потому, что в обработчике прерываний после первого приема сигнала происходит восстановление диспозиции сигналов. Аналогичная ситуация была бы при двукратной отправке процессу сигнала SIGUSR1.

Повторим эксперимент для другого сигнала. Возьмём сигнал SIGILL. Вместо SIGUSR1 перехватываем и обрабатываем сигнал SIGILL (по умолчанию - несуществующая инструкция). Из теоретических данных знаем, что процессы изолированы друг от друга, в том числе у них свои диспозиции сигналов. Проверим это: запустим программу, представленную 111.с, и новую 112.с. И там и там отправим процессу сигнал SIGILL.

*112.c*

|  |
| --- |
| #include <stdio.h>  #include <signal.h>  #include <unistd.h>  #include <stdlib.h>  static void sigHandler(int sig)  {    printf("Catched signal %s\n", sig == SIGUSR1 ? "SIGUSR1" : "SIGUSR2");    printf("Parent = %d\n", getppid());    // востанавливаем старую диспозицию    signal(sig, SIG\_DFL);  }  int main()  {    printf("\nFather started: pid = %i,ppid = %i\n", getpid(), getppid());    signal(SIGILL, sigHandler);    signal(SIGUSR2, sigHandler);    signal(SIGINT, SIG\_DFL);    signal(SIGCHLD, SIG\_IGN);    int forkRes = fork();    if (forkRes == 0)    {      // программа-потомок      printf("\nSon started: pid = %i,ppid = %i\n", getpid(), getppid());        printf("Successfully sent SIGUSR1\n");      return 0;    }    // программа-родитель    wait(NULL);    // ждем сигналов    for (;;)    {      pause();    }    return 0;  } |

Результат:



Вывод: ввиду разной диспозиции сигналов, программа 111.с не обрабатывала сигнал SIGILL написанным обработчиком, а программа 2 – обрабатывала.

Рассмотрим ещё один эксперимент. В программе 113.с узнаем pid процесса, запишем его в файл и перейдём в ожидание сигналов (при этом написан обработчик для SIGUSR2). Запустим её, тем временем в программе 114.с откроется файл, куда раннее записали pid процесса программы 3, соберётся строка и произведётся системный вызов, в результате которого программе 113.с пошлётся сигнал SIGUSR2.

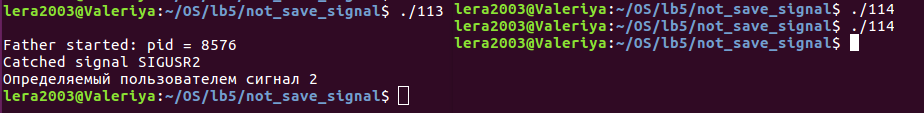
*113.с*

|  |
| --- |
| #include <stdio.h>  #include <signal.h>  #include <unistd.h>  #include <stdlib.h>  static void sigHandler(int sig)  {    printf("Catched signal %s\n", sig == SIGUSR1 ? "SIGUSR1" : "SIGUSR2");    printf("%d", getppid());    // востанавливаем старую диспозицию    signal(sig, SIG\_DFL);  }  int main()  {    signal(SIGUSR2, sigHandler);    FILE\* fp;    fp = fopen("pid\_file.txt", "w");    char str[100];    if (fp == NULL){      perror("FILE");    exit(1);    }    int father\_pid = getpid();    sprintf(str, "%d", father\_pid);    printf("\nFather started: pid = %i\n", father\_pid);    fprintf(fp, str);    fclose(fp);    // ждем сигналов    for (;;)    {      pause();    }      return 0;  } |

*114.с*

|  |
| --- |
| #include <stdio.h>  #include <signal.h>  #include <unistd.h>  #include <stdlib.h>  #include <string.h>  static void sigHandler(int sig)  {    printf("Catched signal %s\n", sig == SIGUSR1 ? "SIGUSR1" : "SIGUSR2");    printf("%d", getppid());    // востанавливаем старую диспозицию   // signal(sig, SIG\_DFL);  }  char\* concat(const char \*s1, const char \*s2){    char\* result = malloc(strlen(s1) + strlen(s2) + 1);    strcpy(result, s1);    strcat(result, s2);    return result;  }  int main()  {    FILE\* fp;    fp = fopen("pid\_file.txt", "r");    char str[100];    if (fp == NULL){      perror("FILE");    exit(1);    }    fgets(str, 100, fp);    //printf("%s\n", str);    char\* s = concat("kill -12 ", str);  //12 == SIGUSR2   // printf("string: [%s]\n", s);    system(s);    fclose(fp);    free(s);    // ждем сигналов      return 0;  } |

Результат:



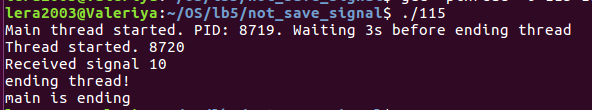
Вывод: проведен эксперимент для процессов, порождаемых в разных файлах. Сигнал обрабатывается точно так же, как и в случае с родственными процессами (т. е. сначала сигнал обрабатывается sighandler, потом восстанавливается диспозиция и программа завершается при второй посылке сигнала).

Задание: Рассмотрим обработку сигналов потоком одного процесса. Используем команду sigwait(sigset\_t, sig). Она прекращает исполнение потока, пока один из сигналов из списка sigset\_t не становится в ожидании (или не послан напрямую), и возвращает номер сигнала в sig. Обработаем SIGUSR1 и SIGUSR2 так: на посыл SIGUSR1 остановим поток, а SIGUSR2 проигнорируем.

*115.с*

|  |
| --- |
| #include <stdio.h>  #include <signal.h>  #include <unistd.h>  #include <stdlib.h>  #include <wait.h>  #include <sys/types.h>  #include <sys/syscall.h>  #include <string.h>  #include <pthread.h>  #define \_GNU\_SOURCE  pthread\_t thread1;  sigset\_t set;  int thread\_id;  void \*thread\_func(void \*args) {      thread\_id = syscall(SYS\_gettid);      printf("Thread started. %d\n", thread\_id);      while(1) {          int signum;          sigwait(&set, &signum);          printf("Received signal %d\n", signum);      if (signum == SIGUSR1){          break;      }      }      sleep(2);      printf("ending thread!\n");      pthread\_exit(NULL);  }  void sig\_handler(int signum){  }  int main() {      signal(SIGUSR1, sig\_handler);      signal(SIGUSR2, SIG\_IGN);      sigemptyset(&set);      sigaddset(&set, SIGTERM);      sigaddset(&set, SIGINT);      sigaddset(&set, SIGUSR1);      if(pthread\_create(&thread1, NULL, thread\_func, NULL)) {          printf("Error creating thread\n");          return 1;      }      printf("Main thread started. PID: %d. Waiting 3s before ending thread\n", getpid());      pthread\_kill(thread1, SIGUSR2);      sleep(3);      pthread\_kill(thread1, SIGUSR1);      pthread\_join(thread1, NULL);      printf("main is ending\n");      return 0;  } |

Результат:



Вывод: сигналы обрабатываются согласно определенными нами обработчиками, а на посыл SIGUSR1 (сигнал 10) поток завершает свою работу.

В отличие от работы с процессами, в случае с потоками нельзя завершить работу одного потока из другой программы с помощью pthread\_kill(pthread\_t), по причине того, что pthread\_t хранится в другом адресном пространстве. Это можно обойти посредством посылки сигнала из одной программы во вторую. Программа create.c создаёт поток и ожидает его завершения, а программа send.c отправит потоку сигнал SIGUSR1 (в результате которого поток и после него процесс завершатся).

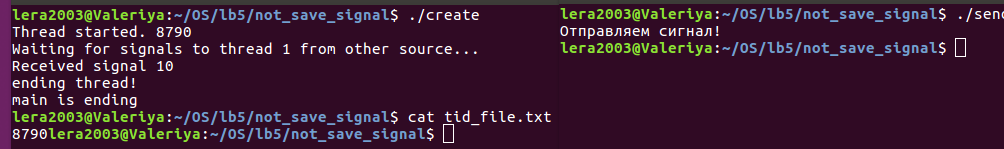
*create.c*

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <signal.h>  #include <pthread.h>  #include <unistd.h>  #include <sys/syscall.h>  #define \_GNU\_SOURCE  pthread\_t thread1;  sigset\_t set;  int thread\_id;  void \*thread\_func(void \*args) {      thread\_id = syscall(SYS\_gettid);      printf("Thread started. %d\n", thread\_id);      while(1) {          int signum;          sigwait(&set, &signum);          printf("Received signal %d\n", signum);      if (signum == SIGUSR1){          break;      }      }      sleep(2);      printf("ending thread!\n");      pthread\_exit(NULL);  }  void sig\_handler(int signum){  }  int main() {      signal(SIGUSR1, sig\_handler);      signal(SIGUSR2, SIG\_IGN);      sigemptyset(&set);      sigaddset(&set, SIGTERM);      sigaddset(&set, SIGINT);      sigaddset(&set, SIGUSR1);      if(pthread\_create(&thread1, NULL, thread\_func, NULL)) {          printf("Error creating thread\n");          return 1;      }      FILE\* fp;      fp = fopen("tid\_file.txt", "w");      char str[100];      if (fp == NULL){          perror("FILE");          exit(1);      }      sprintf(str, "%d", thread\_id);      fprintf(fp, str);      fclose(fp);      printf("Waiting for signals to thread 1 from other source...\n");      pthread\_join(thread1, NULL);      printf("main is ending\n");      return 0;  } |

*send.c*

|  |
| --- |
| #include <stdio.h>  #include <signal.h>  #include <unistd.h>  #include <stdlib.h>  #include <string.h>  // static void sigHandler(int sig)  // {  //   printf("Catched signal %s\n", sig == SIGUSR1 ? "SIGUSR1" : "SIGUSR2");  //   printf("%d", getppid());  // }  char\* concat(const char \*s1, const char \*s2){    char\* result = malloc(strlen(s1) + strlen(s2) + 1);    strcpy(result, s1);    strcat(result, s2);    return result;  }  int main()  {    FILE\* fp;    fp = fopen("tid\_file.txt", "r");    char str[100];    if (fp == NULL){      perror("FILE");    exit(1);    }    fgets(str, 100, fp);    //printf("%s\n", str);    char\* s = concat("kill -10 ", str);  //10 == SIGUSR2   // printf("string: [%s]\n", s);    printf("Отправляем сигнал!\n");    system(s);    fclose(fp);    free(s);    // ждем сигналов      return 0;  } |

Результат:



Вывод: поток принял сигнал, обработал его (завершился) и после него завершился основной процесс.

* 1. **Надежные сигналы.**

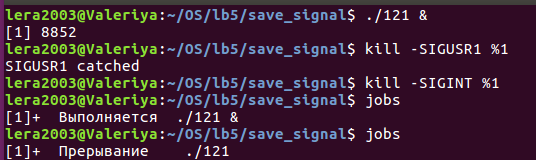
Задание: Создадим программу, позволяющую продемонстрировать возможность отложенной обработки (временного блокирования) сигнала SIGINT.

Вся необходимая для управления сигналами информация передается через указатель на структуру sigaction. Блокировку реализуем, вызвав "засыпание" процесса на одну минуту из обработчика пользовательских сигналов. В основной программе установим диспозицию этих сигналов. С рабочего терминала отправим процессу sigact сигнал SIGUSR1 или SIGUSR2, а затем сигнал SIGINT.

*121.с*

|  |
| --- |
| #include <stdio.h>  #include <signal.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <unistd.h>  #include <stdlib.h>  #include <fcntl.h>  void (\*mysig(int sig, void (\*hnd)(int)))(int)  {      //надежная обработка сигналов      struct sigaction act, oldact;      //задает тип действий процесса      act.sa\_handler = hnd;      //инициализирует набор сигналов, указанный в set, и "очищает" его от всех сигналов      //задает маску сигналов, которые должны блокироваться при обработке сигнала      sigemptyset(&act.sa\_mask);      //добавляют сигналы signum к set      sigaddset(&act.sa\_mask, SIGINT);      //содержит набор флагов, которые могут влиять на поведение процесса при обработке сигнала      act.sa\_flags = 0;      //используется для изменения действий процесса при получении соответствующего сигнала      if (sigaction(sig, &act, 0) < 0)          return SIG\_ERR;      return act.sa\_handler;  }  void hndUSR1(int sig)  {      if (sig != SIGUSR1)      {          printf("Catched bad signal %d\n", sig);          return;      }      printf("SIGUSR1 catched\n");      sleep(10);  }  int main()  {      mysig(SIGUSR1, hndUSR1);      for (;;)      {          pause();      }      return 0;  } |

Результат:



Вывод: сигнал SIGUSR1 принят корректно, но после отсылки SIGINT программа работала ещё 10 секунд, и только после этого завершилась. В этом отличие надежной обработки сигналов от ненадежной: есть возможность отложить прием некоторых других сигналов. Отложенные таким образом сигналы записываются в маску PENDING и обрабатываются после завершения обработки сигналов, которые отложили обработку. Механизм ненадёжных сигналов не позволяет откладывать обработку других сигналов (можно лишь установить игнорирование некоторых сигналов на время обработки).

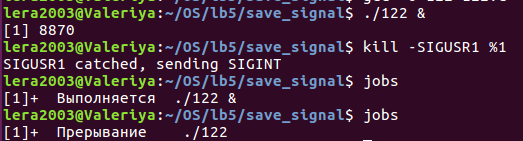
Изменим обработчик сигнала так, чтобы из него производилась отправка другого сигнала.

Пусть из обработчика сигнала SIGUSR1 функцией kill() генерируется сигнал SIGINT. Проанализируем наличие и очередность обработки сигналов.

*122.с*

|  |
| --- |
| #include <stdio.h>  #include <signal.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <unistd.h>  #include <stdlib.h>  #include <fcntl.h>  void (\*mysig(int sig,void (\*hnd)(int)))(int) {  // надежная обработка сигналов      struct sigaction act,oldact;      act.sa\_handler = hnd;      sigemptyset(&act.sa\_mask);      sigaddset(&act.sa\_mask,SIGINT);      act.sa\_flags = 0;      if(sigaction(sig,&act,0) < 0)          return SIG\_ERR;      return act.sa\_handler;  }  void hndUSR1(int sig) {      if(sig != SIGUSR1) {          printf("Catched bad signal %d\n",sig);          return;      }      printf("SIGUSR1 catched, sending SIGINT\n");      kill(getpid(),SIGINT);      sleep(10);  }  int main() {      mysig(SIGUSR1,hndUSR1);      for(;;) {          pause();      }      return 0;  } |

Результат:



Вывод: при генерации сигнала (в данном случае SIGINT) из обработчика другого сигнала обработка сгенерированного сигнала задерживается до конца выполнения текущего обработчика (как и в предыдущем эксперименте).

* 1. **Сигналы POSIX реального времени.**

Некоторые реализации POSIX ОС могут обрабатывать все сигналы как сигналы реального времени, но для UNIX-подобных ОС это не является обязательным. Если мы хотим, чтобы сигналы гарантированно обрабатывались как сигналы реального времени, мы должны:

• использовать сигналы с номерами в диапазоне от SIGRTMIN до SIGRTMAX

• должны указать флаг SA\_SIGINFO при вызове sigaction() с установкой обработчика сигнала

• обработчик сигнала реального времени, устанавливаемый с флагом SA\_SIGINFO, объявляется как:

void func(int signo, siginfo\_t \*info, void \*context); где

signo— номер сигнала,

siginfo\_t — структура, определяемая как typedef struct {

int si\_signo; /\* то же, что и signo \*/

int si\_code; /\* SI\_{USER,QUEUE,TIMER,ASYNCIO,MESGQ} \*/ union sigval si\_value; /\* целое или указатель от отправителя \*/

} siginfo\_t;

на что указывает context — зависит от реализации.

Таким образом, сигналы реального времени несут больше информации, чем прочие сигналы (при отправке сигнала, не обрабатываемого как сигнал реального времени, единственным аргументом обработчика является номер сигнала).

• SIGRTMIN и SIGRTMAX – это еще и макросы (вызывающие sysconf), которые позволяют изменять сами эти значения.

«Характеристики сигналов реального времени» означает следующее:

• Сигналы помещаются в очередь.

• Если сигнал будет порожден несколько раз, он будет несколько раз получен адресатом. Более того, повторения одного и того же сигнала доставляются в порядке очереди (FIFO). Если же сигналы в очередь не помещаются, неоднократно порожденный сигнал будет получен лишь один раз.

• Когда в очередь помещается множество неблокируемых сигналов в диапазоне SIGRTMIN—SIGRTMAX, сигналы с меньшими номерами доставляются раньше сигналов с большими номерами. То есть сигнал с номером SIGRTMIN имеет «больший приоритет», чем сигнал с номером SIGRTMIN+1, и т.д.

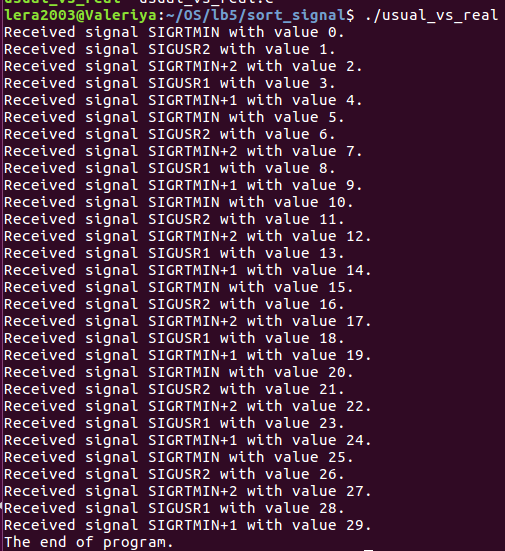
Задание: Проведем эксперимент, позволяющий определить возможность организации очереди для различных типов сигналов, обычных и реального времени.

Из обычных сигналов используем следующие: SIGUSR1 и SIGUSR2. Из сигналов реального времени: SIGRTMIN, SIGRTMIN+1, SIGRTMIN+2. В главной функции в цикле 12 раз процессу отправляется сигнал SIGRTMIN и SIGUSR1 по очереди. В обработчике сигналов SIGRTMIN отправляет сигнал SIGUSR2, который, в свою очередь, отправляет сигнал SIGRTMIN+2. SIGUSR1 же в обработчике вызывает сигнал SIGRTMIN+1.

*usual\_vs\_real.c*

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <signal.h>  #include <unistd.h>  static int reg\_flag = 0;  static int rt\_flag = 0;  static int signal\_counter = 0;  const char\* switch\_int\_to\_sigstr(int sig){      if (sig == SIGRTMIN){          return "SIGRTMIN";      }      else if (sig == SIGRTMIN+1){          return "SIGRTMIN+1";      }      else if (sig == SIGRTMIN+2){          return "SIGRTMIN+2";      }      else if (sig == SIGUSR1){          return "SIGUSR1";      }      else if (sig == SIGUSR2){          return "SIGUSR2";      }      return "UNKNOWN";  }  void handler(const int sig, siginfo\_t\* si, void\* ucontext){      printf("Received signal %s with value %d.\n", switch\_int\_to\_sigstr(sig), si->si\_value.sival\_int);      int pid = getpid();      if (sig == SIGRTMIN){          sigqueue(pid, SIGUSR2, (union sigval){.sival\_int = signal\_counter++});      }      else if (sig == SIGUSR1){          sigqueue(pid, SIGRTMIN+1, (union sigval){.sival\_int = signal\_counter++});      }      if (sig == SIGUSR2){          sigqueue(pid, SIGRTMIN+2, (union sigval){.sival\_int = signal\_counter++});      }  }  void (\*mysig(int sig, void (\*hnd)(int, siginfo\_t\*, void\*)))(int, siginfo\_t\*, void\*){      //обработка сигнала надёжная      struct sigaction act, oldact;      act.sa\_sigaction = hnd;      sigemptyset(&act.sa\_mask);      sigaddset(&act.sa\_mask, SIGINT);      act.sa\_flags = SA\_SIGINFO;      if (sigaction(sig, &act, &oldact) < 0){          perror("sigaction");          exit(-1);      }      return oldact.sa\_sigaction;  }  int main(void){      mysig(SIGUSR1, handler);      mysig(SIGUSR2, handler);      mysig(SIGRTMIN, handler);      mysig(SIGRTMIN+1, handler);      mysig(SIGRTMIN+2, handler);      int pid = getpid();      int signal;      for (int i = 0; i < 12; i++){          signal = (i%2==0) ? SIGRTMIN : SIGUSR1;          sigqueue(pid, signal, (union sigval){.sival\_int = signal\_counter++});      }      sleep(5);      printf("The end of program.\n");      return EXIT\_SUCCESS;  } |

Результат работы программы:



Вывод: Как видно из вывода программы, очередь - цепочка вложенных вызовов разных типов сигналов сохраняется, следовательно, такую очередь организовать можно.

Задание: эксперементально подтвердим, что обработка равноприоритетных сигналов реального времени происходит в порядке FIFO.

Для этого напишем программу, в которой обработчик сигналов будет выводить номер сигнала, его порядковый номер (будем использовать SIGRTMIN). В головной процедуре программа ожидает сигналов извне. Для посылки сигналов напишем скрипт, который 10 раз отправит сигнал SIGRTMIN.

*process\_real.c*

|  |
| --- |
| #include <signal.h>  #include <stdio.h>  #include <unistd.h>  #include <stdlib.h>  int received\_signals[10];  int received\_signals\_value[10];  int received\_signals\_count = 0;  void real\_time\_handler(int sig\_number, siginfo\_t \* info,                         void \* arg \_\_attribute\_\_ ((unused)))  {      received\_signals[received\_signals\_count] = sig\_number - SIGRTMIN;      received\_signals\_value[received\_signals\_count] = info->si\_value.sival\_int;      ++received\_signals\_count;  }  void send\_real\_time\_signal(int sig\_number, int value)  {      union sigval sig\_value;      printf("Sending signal SIRTMIN+%d, value %d\n", sig\_number, value);      sig\_value.sival\_int = value;      if (sigqueue(getpid(), sig\_number + SIGRTMIN, sig\_value) < 0) {          perror("sigqueue");          exit(EXIT\_FAILURE);      }  }  int main()  {      struct sigaction action;      sigset\_t set;      int i;      // Handler setup      action.sa\_sigaction = real\_time\_handler;      sigemptyset(&action.sa\_mask);      action.sa\_flags = SA\_SIGINFO;     sigaddset(&action.sa\_mask, SIGRTMIN + 1);     sigaddset(&action.sa\_mask, SIGRTMIN + 2);     sigaddset(&action.sa\_mask, SIGRTMIN + 3);      // Block all signals      sigprocmask(SIG\_BLOCK, &action.sa\_mask, NULL);      if ((sigaction(SIGRTMIN + 1, &action, NULL) < 0)       || (sigaction(SIGRTMIN + 2, &action, NULL) < 0)       || (sigaction(SIGRTMIN + 3, &action, NULL) < 0)) {          perror("sigaction");          exit(EXIT\_FAILURE);      }      send\_real\_time\_signal(1, 0);      send\_real\_time\_signal(2, 1);      send\_real\_time\_signal(3, 2);      send\_real\_time\_signal(1, 3);      send\_real\_time\_signal(2, 4);      send\_real\_time\_signal(3, 5);      send\_real\_time\_signal(3, 6);      send\_real\_time\_signal(2, 7);      send\_real\_time\_signal(1, 8);      send\_real\_time\_signal(3, 9);      // Unblock all signals      // To make sure we're handling all signals before resuming      sleep(1);      sigfillset(&set);      sigprocmask(SIG\_UNBLOCK, &set, NULL);      // Display results      for (i = 0; i < received\_signals\_count; ++i) {          printf("Received signal SIGRTMIN+%d, value %d\n",                 received\_signals[i], received\_signals\_value[i]);      }      return EXIT\_SUCCESS;  } |

Результат работы:



Вывод: Таким образом, сигналы реального времени обрабатываются следующим образом: вначале обрабатываются наименьшие сигналы, а они уже между собой в порядке порядке FIFO.

**Каналы.**

Различают два типа каналов анонимные (иначе их называют «программные» или «неименованные») и именованные. Они по-разному реализованы, но доступ к ним организуется одинаково с помощью обычных функций read и write (унифицированный подход по типу файловой модели). Одним из свойств программных каналов и FIFO является то, что данные по ним передаются в виде потоков байтов (аналогично соединению TCP). Деление этого потока на самостоятельные записи полностью предоставляется приложению (в отличие, например, от очередей сообщений, которые автоматически расставляют границы между записями, аналогично тому, как это делается в дейтаграммах UDP).

**Неименованные каналы.**

**2. 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:

#include

int pipe(int fd[2]);

/\* возвращает 0 в случае успешного завершения. –1 – в случае ошибки:\*/

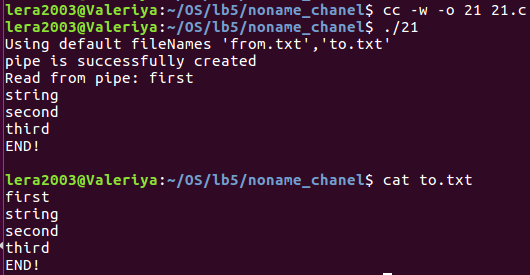
Доступ к дескрипторам канала может получить как процесс, вызвавший pipe() , так и его дочерние процессы. Канал создается одним процессом, может использоваться им единолично (но редко). Как правило, это средство применяется для связи между двумя процессами, следующим образом: процесс создает канал, а затем вызывает fork(), создавая свою копию — дочерний процесс. Затем родительский процесс закрывает открытый для чтения конец канала, а дочерний, в свою очередь, — открытый на запись конец канала. Это обеспечивает одностороннюю передачу данных между процессами. При необходимости передачи данных в обе стороны нужно создавать пару каналов и использовать каждый из них для передачи данных в одну сторону.

Задание: Организуем программу 21.c так, чтобы процесс-родитель создавал неименованный канал, создавал потомка, закрывал канал на запись и записывал в произвольный текстовый файл считываемую из канала информацию. В функции процесса-потомка будет входить считывание данных из файла и запись их в канал.

*21.с*

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <string.h>  #define DEF\_F\_R "from.txt"  #define DEF\_F\_W "to.txt"  int main(int argc, char\*\* argv) {      char fileToRead[32];      char fileToWrite[32];      if (argc < 3) {          printf("Using default fileNames '%s','%s'\n", DEF\_F\_R, DEF\_F\_W);          strcpy(fileToRead, DEF\_F\_R);          strcpy(fileToWrite, DEF\_F\_W);      } else {          strcpy(fileToRead, argv[1]);          strcpy(fileToWrite, argv[2]);      }      int filedes[2];      if (pipe(filedes) < 0) {          printf("Father: can't create pipe\n");          exit(1);      }      printf("pipe is successfully created\n");      if (fork() == 0) {          // процесс сын          // закрывает пайп для чтения          close(filedes[0]);          FILE\* f = fopen(fileToRead, "r");          if (!f) {              printf("Son: cant open file %s\n", fileToRead);              exit(1);          }          char buf[100];          int res;          while (!feof(f)) {              // читаем данные из файла              res = fread(buf, sizeof(char), 100, f);              write(filedes[1], buf, res); // пишем их в пайп          }          fclose(f);          close(filedes[1]);          return 0;      }      // процесс отец      // закрывает пайп для записи      close(filedes[1]);      FILE\* f = fopen(fileToWrite, "w");      if (!f) {          printf("Father: cant open file %s\n", fileToWrite);          exit(1);      }      char buf[100];      int res;      while (1) {          bzero(buf, 100);          res = read(filedes[0], buf, 100);          if (!res)              break;          printf("Read from pipe: %s\n", buf);          fwrite(buf, sizeof(char), res, f);      }      fclose(f);      close(filedes[0]);      return 0;  } |

Результат работы программы:



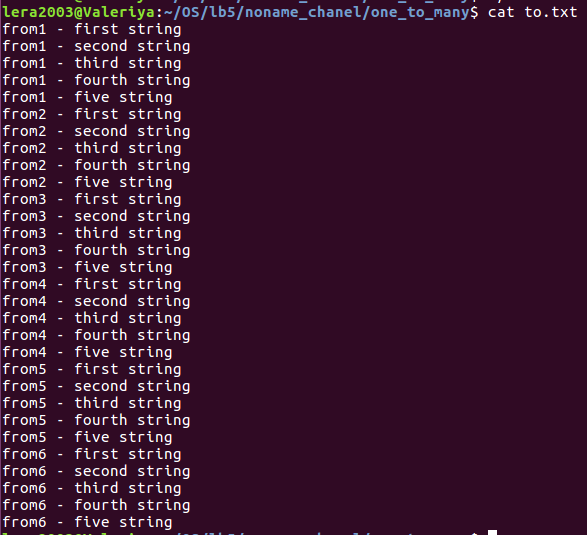
Вывод: содержимое файла from.txt успешно переписалось в изначально пустой файл to.txt с использованием неименованного канала.

Так как процесс-родитель только читает из канала, то дескриптор для записи (filedes[1]) он закрывает, аналогично процесс-сын в начале работы закрывает дескриптор для чтения из канала (filedes[0]).

Задание: Процесс-потомок читает строки из файла, записывает в канал, далее процесс родитель читает из канала строки и записывает их в файл. При этом процесс-потомок закрывает дескриптор канала на чтение. А процесс-родитель закрывает дескриптор канала на запись. Тем самым соблюдается передача данных в одну сторону.

*one\_to\_many.c*

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <string.h>  #define NUM\_CHILDREN 6  int main() {      int i, res;      char buf[100];      int filedes[2\*NUM\_CHILDREN];      for (i = 0; i < NUM\_CHILDREN; i++) {          if (pipe(filedes + i\*2) < 0) {              perror("Can't create pipe");              exit(1);          }          if (fork() == 0) { // Child process              close(filedes[i\*2]); // Close reading end              // Assume each child reads from a different file              // Replace "ChildFileX.txt" with the actual file name              char childFileName[16];              sprintf(childFileName, "from%d.txt", i+1);              FILE\* f = fopen(childFileName, "r");              if (!f) {                  perror("Can't open child file");                  exit(1);              }              while (!feof(f)) {                  res = fread(buf, sizeof(char), 100, f);                  write(filedes[i\*2+1], buf, res); // Write to pipe              }              fclose(f);              close(filedes[i\*2+1]);              exit(0);          }          // Parent process          close(filedes[i\*2+1]); // Close writing end      }      // Parent reads from the pipes and writes to its file      // Replace "ParentFile.txt" with the actual file name      FILE\* f = fopen("to.txt", "w");      if (!f) {          perror("Can't open parent file");          exit(1);      }      for (i = 0; i < NUM\_CHILDREN; i++) {          while (1) {              bzero(buf, 100);              res = read(filedes[i\*2], buf, 100);              if (!res)                  break;              fwrite(buf, sizeof(char), res, f);          }          close(filedes[i\*2]);      }      fclose(f);      return 0;  } |

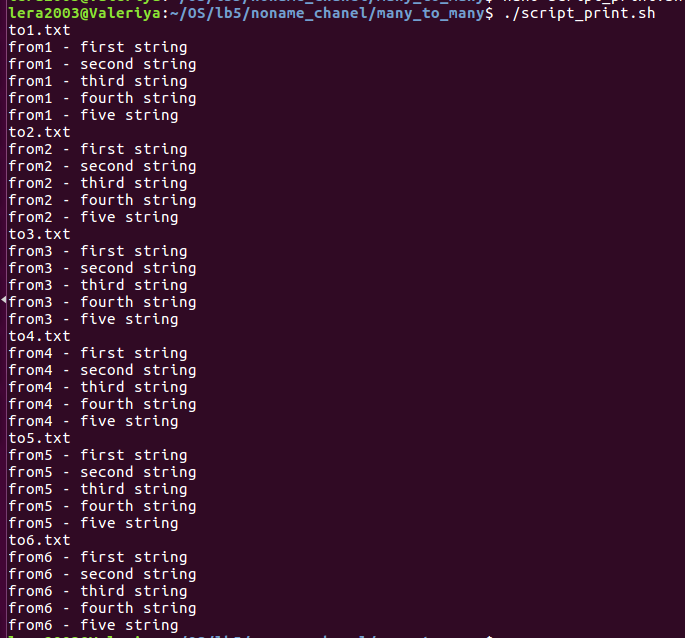


*many\_to\_many.c*

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <string.h>  #define DEF\_F\_R "from.txt"  #define DEF\_F\_W "to.txt"  int main(int argc, char\*\* argv) {      char fileToRead[32];      char fileToWrite[32];      if (argc < 3) {          printf("Using default fileNames '%s','%s'\n", DEF\_F\_R, DEF\_F\_W);          strcpy(fileToRead, DEF\_F\_R);          strcpy(fileToWrite, DEF\_F\_W);      } else {          strcpy(fileToRead, argv[1]);          strcpy(fileToWrite, argv[2]);      }      int filedes[2];      if (pipe(filedes) < 0) {          printf("Father: can't create pipe\n");          exit(1);      }      printf("pipe is successfully created\n");      if (fork() == 0) {          // процесс сын          // закрывает пайп для чтения          close(filedes[0]);          FILE\* f = fopen(fileToRead, "r");          if (!f) {              printf("Son: cant open file %s\n", fileToRead);              exit(1);          }          char buf[100];          int res;          while (!feof(f)) {              // читаем данные из файла              res = fread(buf, sizeof(char), 100, f);              write(filedes[1], buf, res); // пишем их в пайп          }          fclose(f);          close(filedes[1]);          return 0;      }      // процесс отец      // закрывает пайп для записи      close(filedes[1]);      FILE\* f = fopen(fileToWrite, "w");      if (!f) {          printf("Father: cant open file %s\n", fileToWrite);          exit(1);      }      char buf[100];      int res;      while (1) {          bzero(buf, 100);          res = read(filedes[0], buf, 100);          if (!res)              break;          printf("Read from pipe: %s\n", buf);          fwrite(buf, sizeof(char), res, f);      }      fclose(f);      close(filedes[0]);      return 0;  } |

*script.sh*

|  |
| --- |
| #!/bin/bash  for i in {1..5}      do          ./23 "from${i}.txt" "to${i}.txt" &      done      wait |



**3. Именованные каналы.**

Именованные каналы в Unix функционируют подобно неименованным — они позволяют передавать данные только в одну сторону. Однако в отличие от неименованных каналов каждому каналу FIFO сопоставляется полное имя в файловой системе, что позволяет двум неродственным процессам обратиться к одному и тому же FIFO. Аббревиатура FIFO расшифровывается как «first in, first out» — «первым вошел, первым вышел», то есть эти каналы работают как очереди.

После создания канал FIFO должен быть открыт на чтение или запись с помощью либо функции open, либо одной из стандартных функций открытия файлов из библиотеки ввода-вывода (например, fopen). FIFO может быть открыт либо только на чтение, либо только на запись. Нельзя открывать канал на чтение и запись одновременно, поскольку именованные каналы могут быть только односторонними.

Задание: Создать клиент-серверное приложение, демонстрирующее дуплексную (двунаправленную) передачу информации двумя однонаправленными именованными каналами между клиентом и сервером.

В файле server.c в основной программе:

создадим 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). Откроем один канал на запись (chan1), другой - на чтение (chan2) и запустим серверную часть программы.

В серверной части программы:

запишем имя файла в канал 1 (для записи) функцией write(); прочитаем данные из канала 2 и выведем на экран.

В файле client.c запрограммируем функции: открытия каналов для чтения (chan1) и записи (chan2). Из первого канал читается имя файла, во второй канал пишется его содержимое.

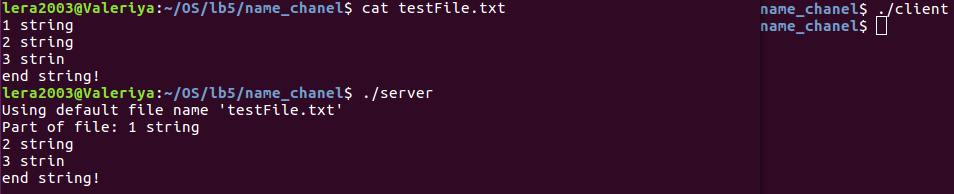
*server.c*

|  |
| --- |
| #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"  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("channel1",S\_IFIFO | 0666,0);      if(res) {          perror("Can't create first channel");          exit(1);}      res = mknod("channel2",S\_IFIFO | 0666,0);      if(res) {          printf("Can't create second channel\n");          exit(1);}  // открываем первый канал для записи      int chan1 = open("channel1",O\_WRONLY);      if(chan1 == -1) {          printf("Can't open channel for writing\n");          exit(0);}  // открываем второй канал для чтения      int chan2 = open("channel2",O\_RDONLY);      if(chan2 == -1) {          printf("Can't open channe2 for reading\n");          exit(0);}  // пишем имя файла в первый канал      write(chan1,fileName,strlen(fileName));  // читаем содержимое файла из второго канала      char buf [100];      for(;;) {          bzero(buf,100);          res = read(chan2,buf,100);          if(res <= 0)              break;          printf("Part of file: %s\n", buf);}      close(chan1);      close(chan2);      unlink("channel1");      unlink("channel2");      return 0;  } |

*client.c*

|  |
| --- |
| #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("channel1", O\_RDONLY);      if (chan1 == -1) {          printf("Can't open channel1 for reading\n");          exit(0);}      int chan2 = open("channel2", O\_WRONLY);      if (chan2 == -1) {          printf("Can't open channel2 for writing\n");          exit(0);}      // читаем имя файла из первого канала      char fileName[100];      bzero(fileName, 100);      int res = read(chan1, fileName, 100);      if (res <= 0) {          printf("Can't read fileName from channel1\n");          exit(0);}      // открываем файл на чтение      FILE \*f = fopen(fileName, "r");      if (!f) {          printf("Can't open file %s\n", fileName);          exit(0);}      // читаем из файла и пишем во второй канал      char buf[100];      while (!feof(f)) {          // читаем данные из файла          res = fread(buf, sizeof(char), 100, f);          // пишем их в канал          write(chan2, buf, res);}      fclose(f);      close(chan1);      close(chan2);      return 0;  } |

Результат:



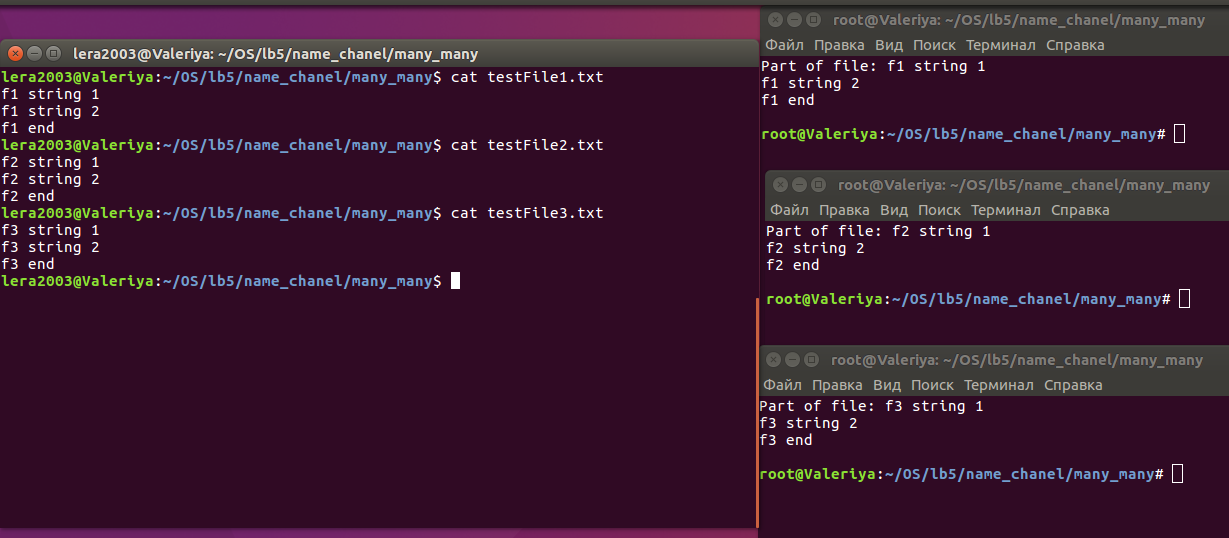
Вывод: сервер создает два канала, записывает в один из них имя файла и ждёт данные от клиента. Каналы создаются в рабочей папке сервера, и использовать их может любой процесс, а не только дочерний по отношению к серверу. Клиент после запуска также открывает уже созданные каналы, считывает имя файла и отсылает серверу его содержимое, используя второй канал. После завершения передачи, сервер уничтожает каналы с помощью функции unlink().

Напишем скрипт, создающий множество клиентов и серверов.

*manys\_manyc.sh*

|  |
| --- |
| #!/bin/bash  gcc server.c -o server  gcc client.c -o client  for i in {1..3}  do      gnome-terminal -- bash -c "./server testFile$i.txt channel${i}1 channel${i}2; exec bash" &      sleep 2      gnome-terminal -- bash -c "./client channel${i}1 channel${i}2; exec bash" &      sleep 1  done |

Результат:



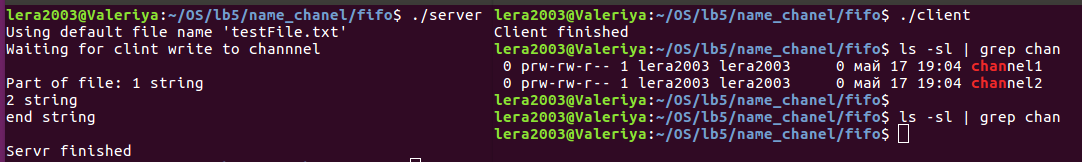
Несмотря на то, что именованные каналы являются отдельным типом файлов и могут быть видимы разными процессами даже в распределенной файловой системе, использование FIFO для взаимодействия удаленных процессов и обмена информацией между ними невозможно. Так, как и в этом случае для передачи данных задействовано ядро. Создаваемый файл служит для получения данных о расположении FIFO в адресном пространстве ядра и его состоянии.

Задание: Продемонстрируем это на примере. Изменим ранее использованную программу так, чтобы сервер, перед тем как читать данные из канала, ожидал ввода пользователя. Исходный код клиента оставим неизменным.

*server.c*

|  |
| --- |
| #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"  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("channel1",S\_IFIFO | 0666,0);      if(res) {          printf("Can't create first channel\n");          exit(1);      }      res = mknod("channel2",S\_IFIFO | 0666,0);      if(res) {          printf("Can't create second channel\n");          exit(1);      }  // открываем первый канал для записи      int chan1 = open("channel1",O\_WRONLY);      if(chan1 == -1) {          printf("Can't open channel for writing\n");          exit(0);      }  // открываем второй канал для чтения      int chan2 = open("channel2",O\_RDONLY);      if(chan2 == -1) {          printf("Can't open channe2 for reading\n");          exit(0);      }  // пишем имя файла в первый канал      write(chan1,fileName,strlen(fileName));  // читаем содержимое файла из второго канала      char buf [100];      printf("Waiting for clint write to channnel\n");      getchar();      for(;;) {          bzero(buf,100);          res = read(chan2,buf,100);          if(res <= 0)              break;          printf("Part of file: %s\n", buf);      }      close(chan1);      close(chan2);      unlink("channel1");      unlink("channel2");      printf("Servr finished\n");      return 0;  } |

Результат:



Вывод: размер файла канала не изменяется, несмотря на записанные данные, это свидетельствует о том, что файл используется не как хранилище пересылаемых данных, а только для получения информации системой о них. Сами данные проходят через ядро ОС. Буква p в строке prw-rw-r—определяет, что файл является каналом.

**4. 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).

Следует заметить, что, к сожалению, не определены системные вызовы, которые позволяют читать сразу из нескольких очередей сообщений, или из очередей сообщений и файловых дескрипторов. Видимо, отчасти и поэтому очереди сообщений широко не используются.

Задание: Создадим 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 со следующей структурой:

void main(void)

{

Message msg\_rcv; //принимаемое сообщение Message msg\_snd; //посылаемое сообщение

key\_t key; //ключ, необходимый для создания очереди int length, n;

signal(SIGINT, sig\_hndlr);

//получение ключа

if((key = ftok("/home/your\_path/test.txt", 'A')) < 0)

//ftok - преобразует имя файла и идентификатор проекта в ключ для системных вызовов (для работы с очередью)

{

printf("Server : can't receive a key\n"); exit(-1);

}

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 client.c 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().

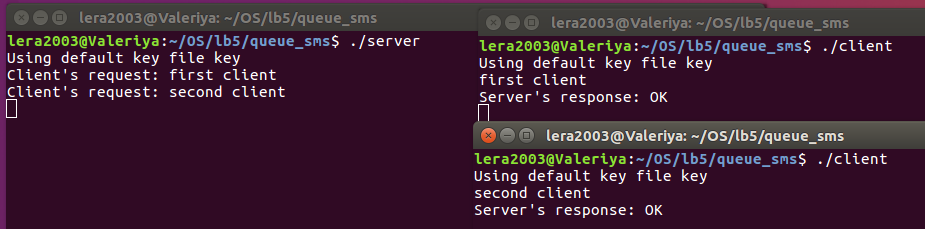
*server.c*

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <sys/ipc.h>  #include <sys/msg.h>  #include <sys/types.h>  #include <signal.h>  #include <unistd.h>  #include <strings.h>  #include <string.h>  #define DEF\_KEY\_FILE "key"  typedef struct { long type; char buf[100];  } Message;  int queue;  void intHandler(int sig) {      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);      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;  key = ftok(keyFile,'Q');  if(key == -1) {  printf("no got key for the key file %s and id 'Q'\n",keyFile);  exit(1);  }  queue = msgget(key,IPC\_CREAT | 0666);  if (queue < 0) {  printf("Can't create queue\n");  exit(4);  }  // до этого момента вызывали exit(), а не kill, т.к. очередь  // еще не была создана  signal(SIGINT,intHandler);  // основной цикл работы сервера  Message mes;  int res;  for(;;) {  bzero(mes.buf,100);  // 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;  } |

*client.c*

|  |
| --- |
| #include <stdio.h>  #include <stdlib.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;      char buf[100];  } Message;  int queue;  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;      key = ftok(keyFile, 'Q');      if (key == -1)      {          printf("no got key for key file %s and id 'Q'\n", keyFile);          exit(1);      }      queue = msgget(key, 0);      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;  } |

Результат:



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 на экран 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, он шлет сообщение с типом 1, содержащее считанные данные и ожидает от сервера подтверждения о принятии.

Модернизируем код и напишем скрипты, которые создают множество серверов и клиентов или множество клиентов для одного сервера.

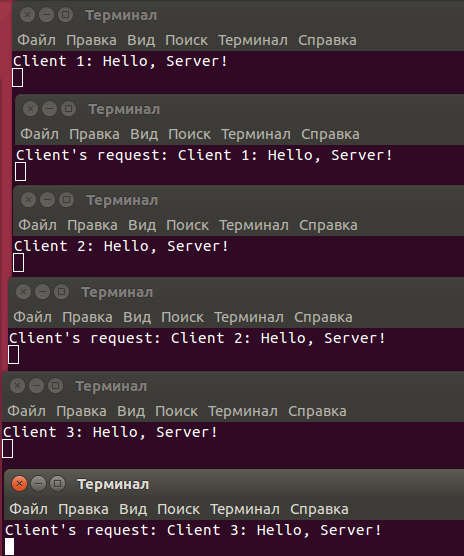
*many\_to\_many.sh*

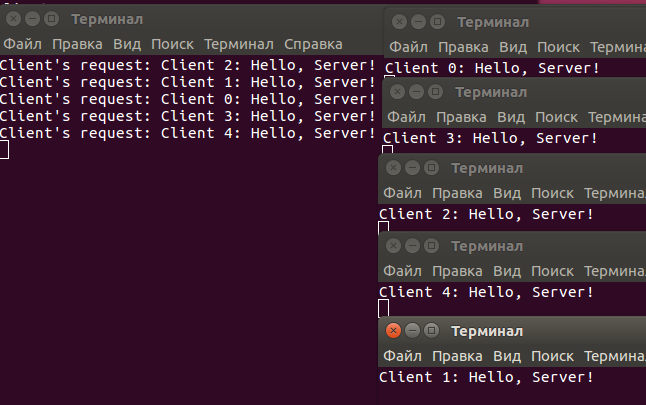
|  |
| --- |
| #!/bin/bash  gcc -o server server.c  gcc -o client client.c  # Создание 3 пар сервер-клиент  for i in {1..3}  do      key=$i      # Запуск сервера      gnome-terminal -- bash -c "./server $key; exec bash"      sleep 1      # Запуск клиента      gnome-terminal -- bash -c "./client $key $i; exec bash"      sleep 1  done |

*many\_to\_one.sh*

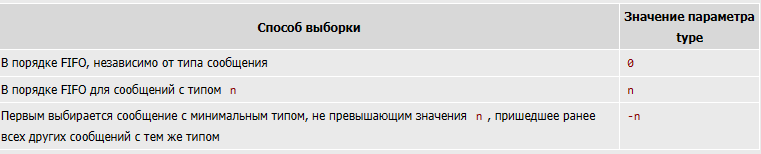
|  |
| --- |
| #!/bin/bash  # Запускаем сервер с очередью 9000  gnome-terminal -- ./server 9000 &  echo "Started server on queue"  # Даем серверу время на запуск  sleep 2  # Запускаем 5 клиентов, каждый подключается к серверу  for i in {0..4}; do      gnome-terminal -- ./client 9000 $i &      echo "Started client $i connecting to queue"  done |

Результат:





Задание: реализуем фильтрацию сообщений. Добавим в файле server.c проверку на тип сообщения. Будем блокировать все сообщения с типом 5L. Создадим 3 клиента: два с допустимым типом сообщения, один с недопустимым.



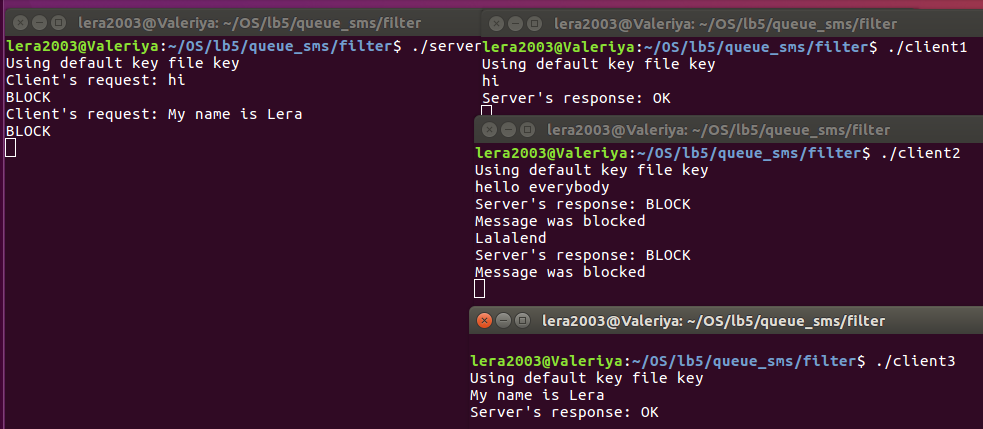
*server.c*

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <sys/ipc.h>  #include <sys/msg.h>  #include <sys/types.h>  #include <signal.h>  #include <unistd.h>  #include <strings.h>  #include <string.h>  #define DEF\_KEY\_FILE "key"  typedef struct  {      long type;      char buf[100];  } Message;  int queue;  void intHandler(int sig)  {      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);      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;      key = ftok(keyFile, 'Q');      if (key == -1)      {          printf("no got key for the key file %s and id 'Q'\n", keyFile);          exit(1);      }      queue = msgget(key, IPC\_CREAT | 0666);      if (queue < 0)      {          printf("Can't create queue\n");          exit(4);      }      // до этого момента вызывали exit(), а не kill, т.к. очередь      // еще не была создана      signal(SIGINT, intHandler);      // основной цикл работы сервера      Message mes;      int res;      for (;;)      {          bzero(mes.buf, 100);          // получаем первое сообщение с типом 1          res = msgrcv(queue, &mes, sizeof(Message), 0, 0);          if (res < 0)          {              printf("Error while recving msg\n");              kill(getpid(), SIGINT);          }          if (mes.type == 5L) {              printf("BLOCK\n");              mes.type = 10L;              bzero(mes.buf, 100);              strcpy(mes.buf, "BLOCK");              res = msgsnd(queue, (void \*)&mes, sizeof(Message), 0);              if (res != 0)              {                  printf("error while sending msg\n");                  kill(getpid(), SIGINT);              }              continue;          }          else {              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;  } |

*client1|2|3.c*

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <sys/ipc.h>  #include <sys/msg.h>  #include <sys/types.h>  #include <signal.h>  #include <string.h>  #define DEF\_KEY\_FILE "key"  typedef struct  {      long type;      char buf[100];  } Message;  int queue;  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;      key = ftok(keyFile, 'Q');      if (key == -1)      {          printf("no got key for key file %s and id 'Q'\n", keyFile);          exit(1);      }      queue = msgget(key, 0);      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 = 9L;          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), 0, 0);          if (res < 0)          {              printf("Error while recving msg\n");              exit(1);          }          printf("Server's response: %s\n", mes.buf);          if (mes.type != 2L) {              printf("Message was blocked\n");              continue;          }      }      return 0;  } |

Результат работы программы:

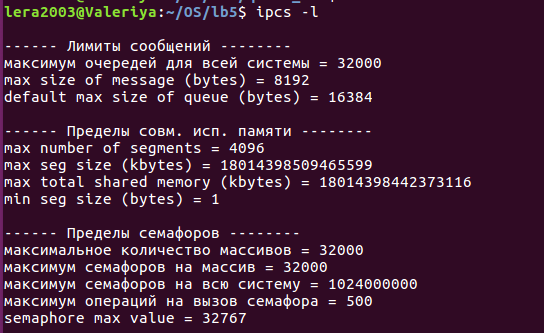


Выводы: таким образом, по значению типа сообщения можно произвести фильтрацию. Например, блокировать сообщенич с определенным типом и посылать отчет об этом клиенту. Тип сообщения может принимать любое целое значение. Для определения типа данный параметр хранится в структуре Message, что позволяет произвести фильтрацию сервером получаемых сообщений.

Количественные ограничения средств IPC.

Максимальные и минимальные значения констант можно выяснить различными способами, в частности, просматривая соответствующие файлы каталога /proc/sys/kernel.

Наиболее простой способ – воспользоваться утилитой ipcs с ключом -l.



Из скриншота видно, что, например, размер одного сообщения не может быть больше 8192 байт, а очередь может содержать не более 32000 сообщений в один момент времени.

**5. Семафоры и разделяемая память.**

**5.1. 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.

Задание: 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.

*server.c*

|  |
| --- |
| #include <stdio.h>  #include <stdlib.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 <string.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);          }      }  } |

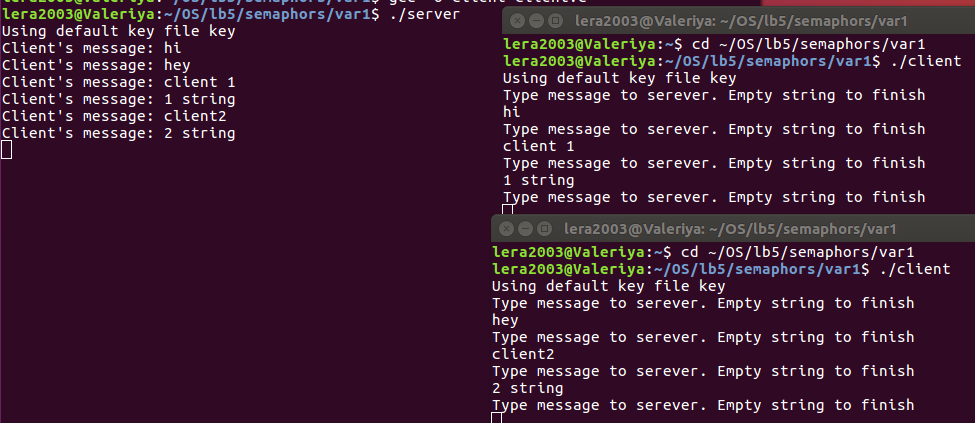
*client.c*

|  |
| --- |
| #include <stdio.h>  #include <stdlib.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 <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);      }  } |

*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}; |

Результат:



Если в изначальной задаче использовать только один семафор для контроля над доступом к общей памяти, то могут возникнуть следующие проблемы:

*Перезапись данных до их чтения:* Если процесс записи в общую память сможет записать новые данные до того, как процесс чтения успеет прочитать предыдущие данные, то данные могут быть потеряны. Это происходит, когда семафор разрешает процессу записи доступ к общей памяти до того, как процесс чтения успел прочитать предыдущие данные.

*Повторное чтение одних и тех же данных:* Если процесс чтения сможет прочитать данные из общей памяти до того, как процесс записи успеет записать новые данные, то процесс чтения может повторно прочитать одни и те же данные. Это происходит, когда семафор разрешает процессу чтения доступ к общей памяти до того, как процесс записи успел записать новые данные.

Эти проблемы возникают из-за отсутствия надлежащего контроля над тем, когда и в каком порядке процессы чтения и записи получают доступ к общей памяти. Для надлежащего контроля обычно требуется использование двух семафоров: одного для контроля доступа процесса записи, и второго - для контроля доступа процесса чтения.

Семафоры, как инструменты синхронизации, используются для контроля доступа к общим ресурсам, в частности, для обеспечения исключающего доступа. Вот несколько примеров ситуаций, в которых может быть достаточно одного семафора:

*Очередь заданий:* Возьмем для примера сервер, который обрабатывает входящие запросы от клиентов. Все эти запросы помещаются в общую очередь. В то же время, у сервера есть несколько рабочих потоков (worker threads), которые забирают задачи из этой очереди и обрабатывают их. Здесь важно обеспечить, чтобы в одно и то же время только один поток мог взять задачу из очереди, чтобы не возникало конфликтов или ошибок. В этом случае можно использовать один семафор для синхронизации доступа к очереди.

*Доступ к общему файлу или ресурсу:* Представьте, что у вас есть несколько потоков или процессов, которые хотят записать данные в один и тот же файл. Если они начнут делать это одновременно, это может привести к проблемам. В этом случае можно использовать семафор, чтобы гарантировать, что только один процесс или поток может записывать в файл в любой момент времени.

*Обновление общих данных:* Предположим, у вас есть общий счетчик, который используется несколькими потоками. Если несколько потоков попытаются увеличить счетчик одновременно, это может привести к "гонкам" (race conditions) и некорректному результату. Один семафор может быть использован для того, чтобы гарантировать, что только один поток обновляет счетчик в любой момент времени.

В каждом из этих примеров один семафор используется для контроля доступа к общему ресурсу и предотвращения проблем, связанных с одновременным доступом.

**5.2. 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.

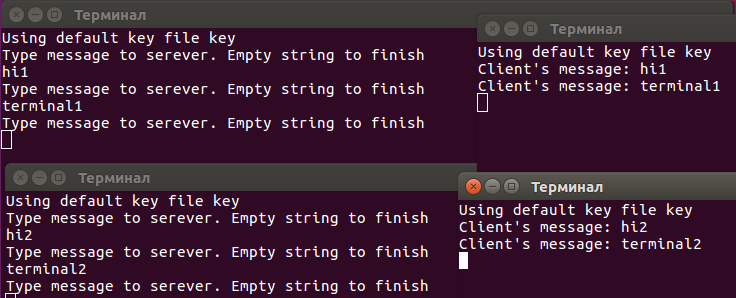
Легко понять, что это условие не приводит к необходимости использования дополнительных средств синхронизации. Теперь вместо одного процесса, за каждый семафор будут конкурировать несколько. Но повторная запись или чтение все также невозможно. Так как, чтобы очередной процесс- писатель отработал, нужно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

Реализовано добавление множества клиентов и серверов для двух семафоров.

*script.sh*

|  |
| --- |
| #!/bin/bash  # Количество читателей и писателей  num\_readers=2  num\_writers=2  # Запуск читателей  for ((i=0; i<$num\_readers; i++)); do      gnome-terminal -- ./server  done  # Запуск писателей  for ((i=0; i<$num\_writers; i++)); do      gnome-terminal -- ./client  done |

Результат:



**5.3. 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.

В качестве разделяемого ресурса используется массив, находящийся в разделяемой памяти. Ячейка памяти, расположенная за последним элементом массива, интерпретируется как индекс последнего записанного элемента.

*client.c*

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <sys/ipc.h>  #include <sys/msg.h>  #include <sys/types.h>  #include <signal.h>  #include <unistd.h>  #include <strings.h>  #include <string.h>  #include <sys/sem.h>  #include <sys/shm.h>  #include "shm.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);  } |

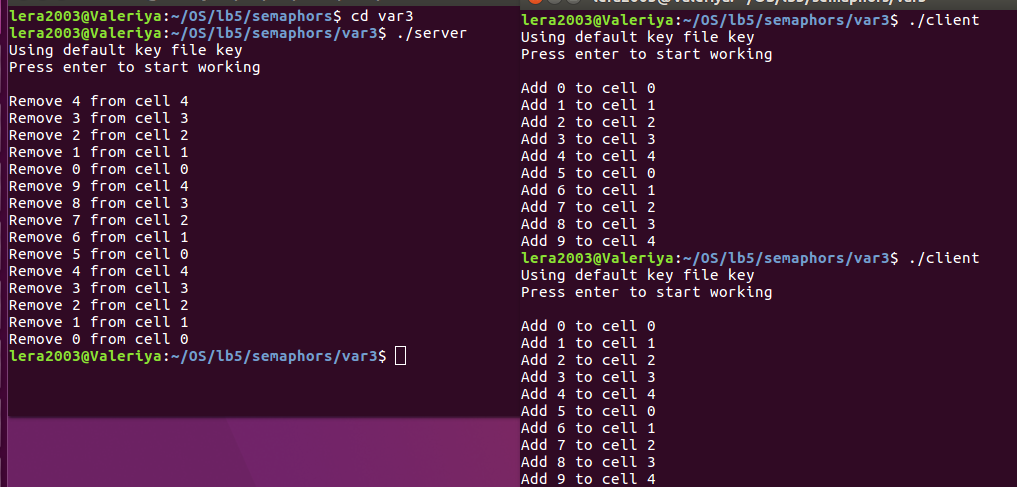
*server.c*

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <sys/ipc.h>  #include <sys/msg.h>  #include <sys/types.h>  #include <signal.h>  #include <unistd.h>  #include <strings.h>  #include <string.h>  #include <sys/ipc.h>  #include <sys/sem.h>  #include <sys/shm.h>  #include <sys/time.h>  #include "shm.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 записывающий клиент 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);   }   }  } |

*shm.h*

|  |
| --- |
| #define DEF\_KEY\_FILE "key"  #define BUF\_SIZE 5  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};  static struct sembuf mem\_lock[1] = {2,-1,0};  static struct sembuf mem\_unlock[1] = {2,1,0};  static struct sembuf setFree[1] = {0,BUF\_SIZE,0};  static struct sembuf waitNotEmpty[1] = {1,-1,0};  static struct sembuf releaseEmpty[1] = {0,1,0};  static struct sembuf waitNotFull[1] = {0,-1,0};  static struct sembuf releaseFull[1] = {1,1,0}; |

Результат работы программы:



Выводы: Процессы-писатели записывают по 10 чисел в массив, процесс-читатель считывает первые 15 из записанных. По результатам проверяем, что синхронизация работает корректно, выхода за пределы массива нет, записанные данные не затираются до их прочтения. К примеру, в нулевую ячейку сначала первый поток-писатель записал 0, потом он же записал 5, а потом второй поток-писатель записал также 0. Проверяем, что все данные были прочитаны.

**Вывод.**

Изучены сигналы, два вида: надежные и ненадежные. Ненадежные сигналы работаю так: если сигнал еще не зарегистрирован, зарегистрируйте его прямо сейчас, если он уже зарегистрирован, ничего не делайте. Надежные: каждый раз, когда надежный сигнал отправляется процессу, независимо от того, получил ли процесс тот же самый сигнал раньше, узел сигнатуры всегда будет добавляться в очередь для обработки. Также изучены сигналы реального времени, которые способны добавлять несколько функций по сравнению с обычными сигналами, например, новый диапазон сигналов для прикладных целей, возможность ставить в очередь ожидающие сигналы и возможность доставлять слово данных с сигналом. У сигналов реального времени приоритет от меньших к большим, а одинаковые между собой обрабатываются в порядке FIFO.

Рассмотрены два вида каналов – именованные и неименованные. Именованный канал может служить для общения и синхронизации произвольных процессов, знающих имя данного программного канала и имеющих соответствующие права доступа. Неименованным программным каналом могут пользоваться только создавший его процесс и его потомки (необязательно прямые). Операционные система позволяет определить, что были созданы каналы, при выводе ls –l в строке, отвечающей за тип файла, первой буквой будет p, что и определяется системой как канал.

Рассмотрена очередь сообщений, которая 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.

Изучены также семафоры и разделяемая память. Семафор – это механизм, который позволяет конкурирующим процессам и потокам работать с общими ресурсами и помогает в решении различных проблем синхронизации. А разделяемая память – это тип межпроцессного взаимодействия, который схематично можно изобразить как некую именованную область в памяти, к которой обращаются одновременно два процесса. Проанализировано на основе эксперементов с помощью написанных программ, что для реализации задачи один процесс выполняет запись в разделяемую память, а другой процесс выполняет чтение из нее, достаточно двух семафоров. Также проведен эксперимент с использованием не единичной памяти, она была представлена в виде массива.

**Список источников.**

1. «Системное программное обеспечение. Межпроцессные взаимодействия в операционных системах. Учебное пособие» Душутина Е.В.
2. Сайт [The Linux Programming Interface (man7.org)](https://man7.org/tlpi/index.html)
3. Сайт [OpenNet: Архив документации: Программирование в Linux](https://www.opennet.ru/docs/137.shtml)
4. Сайт [Ubuntu Manpage: Welcome](https://manpages.ubuntu.com/)
5. Сайт <https://www.opennet.ru/docs/RUS/linux_parallel/node7.html>