**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра МО ЭВМ**

**ОТЧЕТ**

**по лабораторной работе №4**

**по дисциплине «Операционные системы»**

**Тема: Взаимодействие родственных процессов. Управление процессами посредством сигналов. Многонитевое функционирование.**

|  |  |  |
| --- | --- | --- |
| 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:

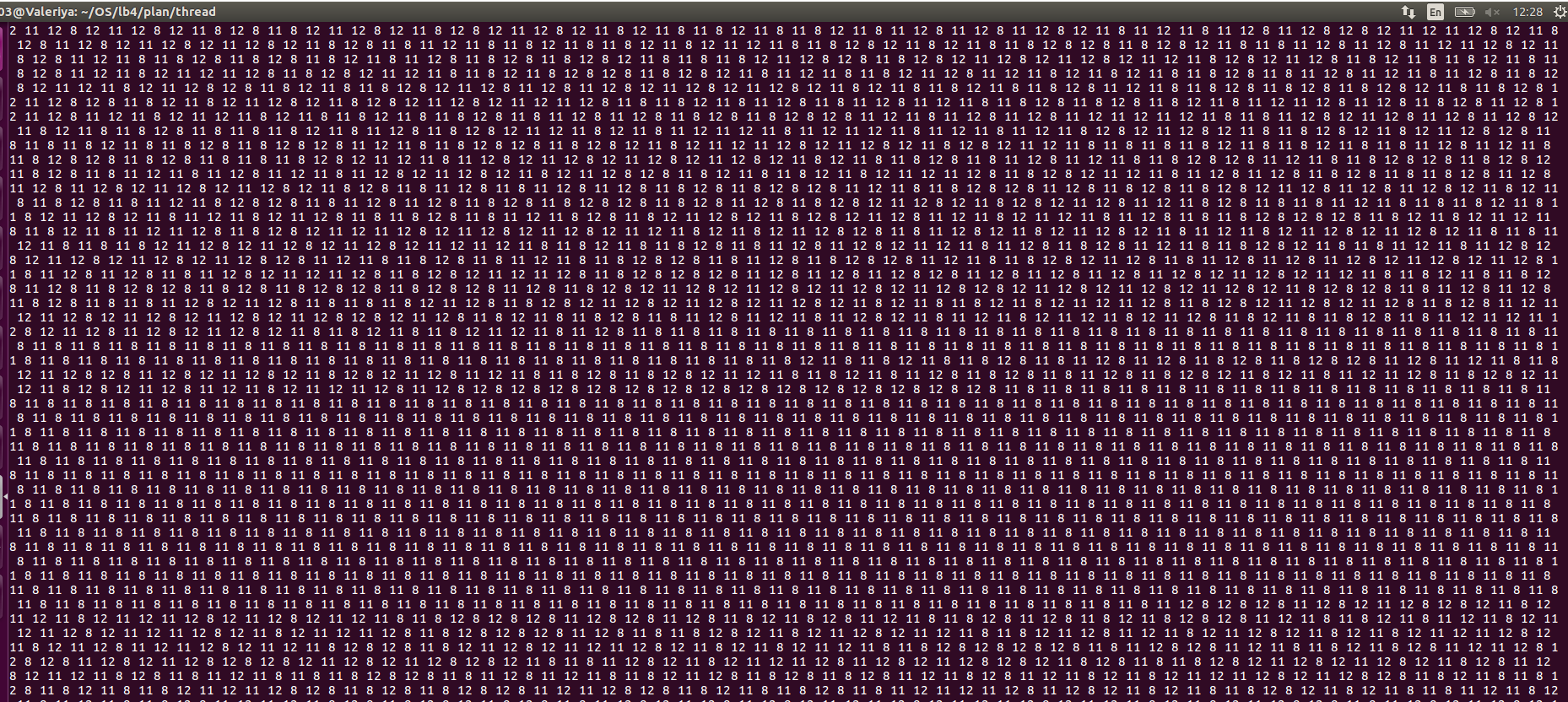
Linux Valeriya 4.15.0-142-generic #146~16.04.1-Ubuntu SMP Tue Apr 13 09:27:15 UTC 2021 x86\_64 x86\_64 x86\_64 GNU/Linux

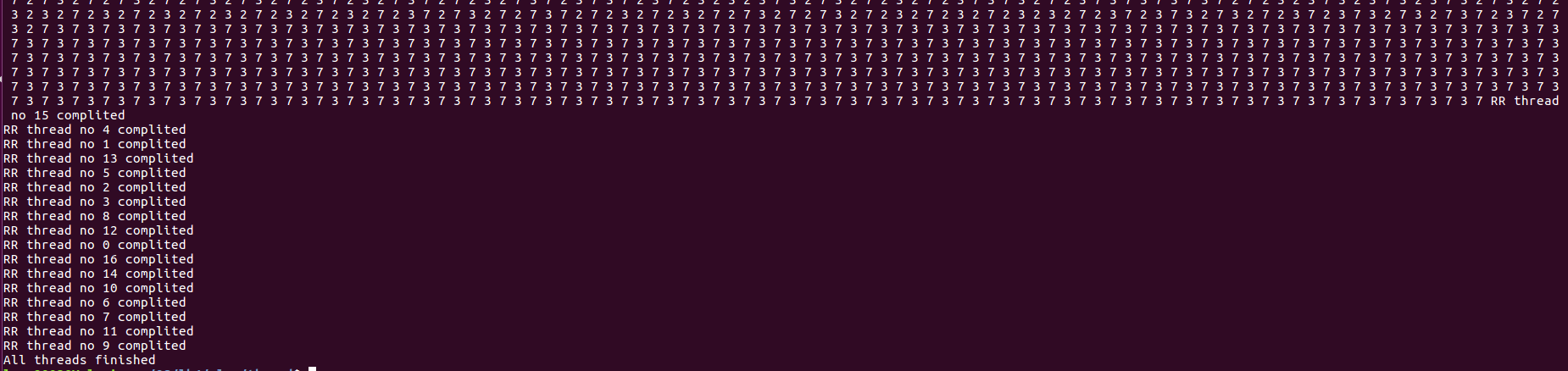
**Проанализируйте процедуру планирования для процессов и потоков одного процесса.**

Пронаблюдаем как процессы сменяют друг друга во время выполнения. Во время работы программы мы в каждом процессе проверяем запись в сохранённые данные, если последним записанным процессом был не нынешний, то мы добавляем нынешний процесс в запись. В конце работы программы все записи смены процессов, а также очерёдность их конца выводятся.

*rr.c*

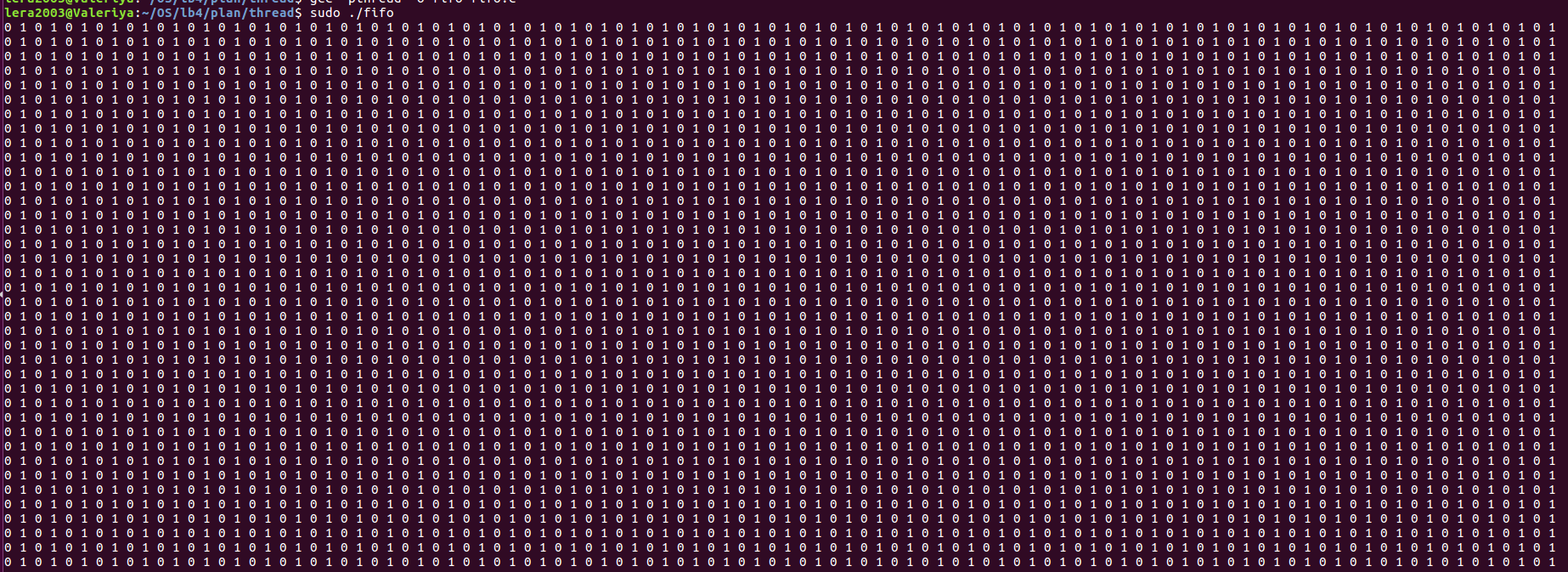
|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <sched.h>  #include <errno.h>  #include <sys/mman.h>  #define THREADS\_AMOUNT 17  #define OPERATIONS\_AMOUNT 10000000  #define OPERATIONS\_CHANGES THREADS\_AMOUNT\*OPERATIONS\_AMOUNT  int \*free\_index;  int \*shared\_memory;  int \*index\_proc;  int \*shared\_memory\_proc;  void\* create\_shared\_memory(size\_t size)  {      int protection = PROT\_READ | PROT\_WRITE;      int visibility = MAP\_SHARED | MAP\_ANONYMOUS;      return mmap(NULL, size, protection, visibility, -1, 0);  }  void \*thread\_fifo\_schedule(void \*attr)  {      for (int i = 0; i < OPERATIONS\_AMOUNT; i++)      {          if (shared\_memory\_proc[(\*index\_proc) - 1] != \*((int\*)attr))          {              shared\_memory\_proc[(\*index\_proc)++] = \*((int\*)attr);          }      }      shared\_memory[(\*free\_index)++] = \*((int\*)attr);  }  void \*thread\_rr\_schedule(void \*attr)  {      for (int i = 0; i < OPERATIONS\_AMOUNT; i++)      {          if (shared\_memory\_proc[(\*index\_proc) - 1] != \*((int\*)attr))          {              shared\_memory\_proc[(\*index\_proc)++] = \*((int\*)attr);          }      }      shared\_memory[(\*free\_index)++] = \*((int\*)attr);  }  int main()  {      free\_index = create\_shared\_memory(sizeof(int));      index\_proc = create\_shared\_memory(sizeof(int));      \*free\_index = 0;      \*index\_proc = 1;      shared\_memory = create\_shared\_memory(sizeof(int) \* THREADS\_AMOUNT);      shared\_memory\_proc = create\_shared\_memory(sizeof(int) \* OPERATIONS\_CHANGES);      for (int i = 0; i < THREADS\_AMOUNT; i++)      {          shared\_memory[i] = -1;      }      for (int i = 0; i < OPERATIONS\_CHANGES; i++)      {          shared\_memory\_proc[i] = -1;      }      pthread\_t threads[THREADS\_AMOUNT];      pthread\_attr\_t threads\_attrs[THREADS\_AMOUNT];      struct sched\_param params[THREADS\_AMOUNT];      int priorityFIFO = 50;      int priorityRR = 50;      int thread\_id[THREADS\_AMOUNT];      for (int i = 0; i < THREADS\_AMOUNT; i++)      {          pthread\_attr\_init(&threads\_attrs[i]);          pthread\_attr\_setinheritsched(&threads\_attrs[i], PTHREAD\_EXPLICIT\_SCHED);          params[i].sched\_priority = priorityRR;          pthread\_attr\_setschedpolicy(&threads\_attrs[i], SCHED\_RR);          pthread\_attr\_setschedparam(&threads\_attrs[i], &params[i]);          thread\_id[i] = i;          if(pthread\_create(&threads[i], &threads\_attrs[i], (i % 2 == 0) ? thread\_rr\_schedule : thread\_fifo\_schedule, &thread\_id[i]))          {              fprintf(stderr, "Error creating thread %d\n", i);          }      }      for (int i = 0; i < THREADS\_AMOUNT; i++)      {          pthread\_join(threads[i], NULL);      }      for (int i = 1; i < OPERATIONS\_CHANGES; i++)      {          if (shared\_memory\_proc[i] == -1)              break;          printf("%d ", shared\_memory\_proc[i]);      }      for (int i = 0; i < THREADS\_AMOUNT; i++)      {          printf("%s thread no %d complited\n", "RR", shared\_memory[i]);      }      printf("All threads finished\n");      return 0;  } |

**

**

*fifo.c*

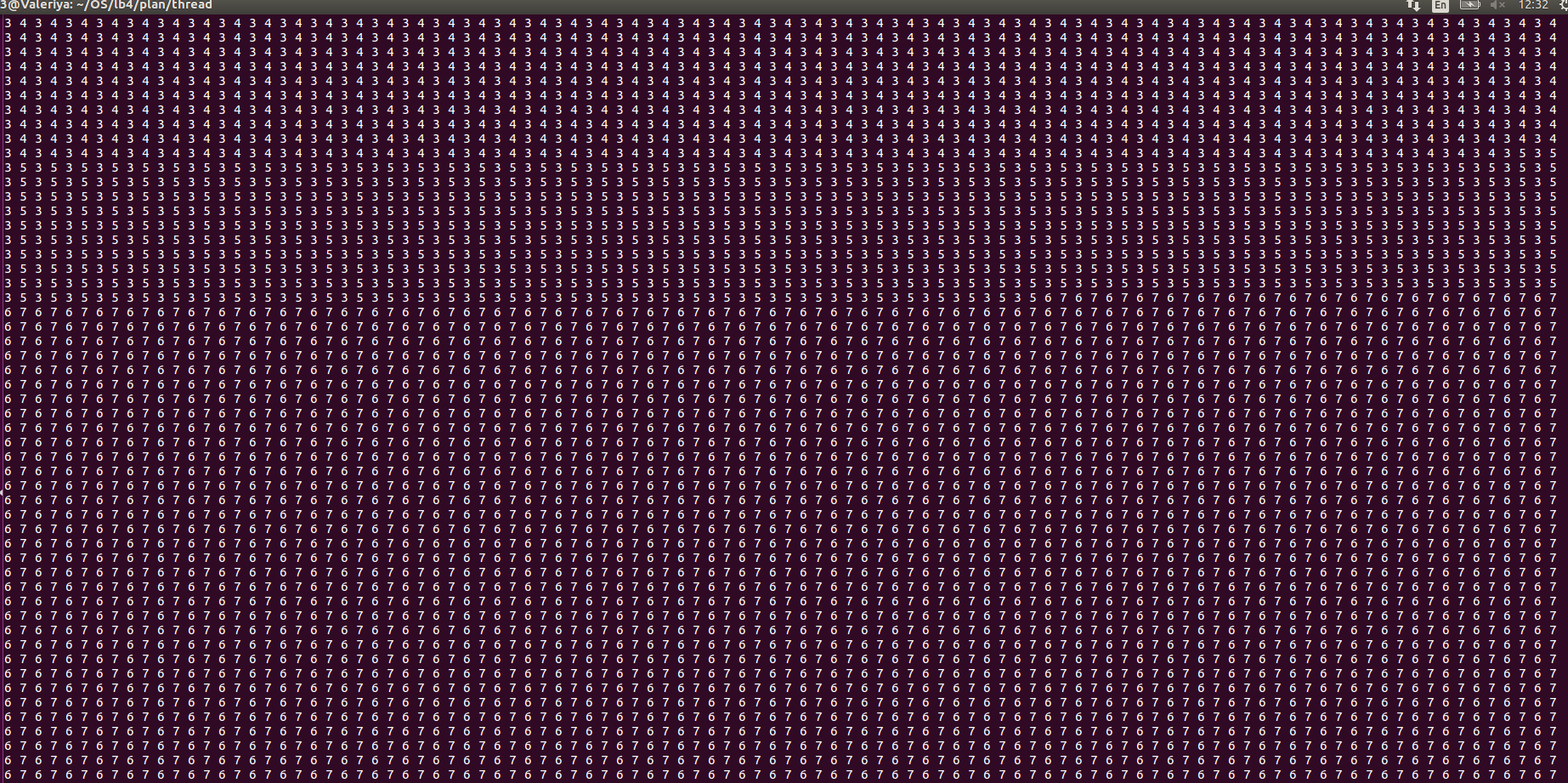
|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <sched.h>  #include <errno.h>  #include <sys/mman.h>  #define THREADS\_AMOUNT 17  #define OPERATIONS\_AMOUNT 100000  #define OPERATIONS\_CHANGES THREADS\_AMOUNT\*OPERATIONS\_AMOUNT  int \*free\_index;  int \*shared\_memory;  int \*index\_proc;  int \*shared\_memory\_proc;  void\* create\_shared\_memory(size\_t size)  {      int protection = PROT\_READ | PROT\_WRITE;      int visibility = MAP\_SHARED | MAP\_ANONYMOUS;      return mmap(NULL, size, protection, visibility, -1, 0);  }  void \*thread\_fifo\_schedule(void \*attr)  {      for (int i = 0; i < OPERATIONS\_AMOUNT; i++)      {          if (shared\_memory\_proc[(\*index\_proc) - 1] != \*((int\*)attr))          {              shared\_memory\_proc[(\*index\_proc)++] = \*((int\*)attr);          }      }      shared\_memory[(\*free\_index)++] = \*((int\*)attr);  }  void \*thread\_rr\_schedule(void \*attr)  {      for (int i = 0; i < OPERATIONS\_AMOUNT; i++)      {          if (shared\_memory\_proc[(\*index\_proc) - 1] != \*((int\*)attr))          {              shared\_memory\_proc[(\*index\_proc)++] = \*((int\*)attr);          }      }      shared\_memory[(\*free\_index)++] = \*((int\*)attr);  }  int main()  {      free\_index = create\_shared\_memory(sizeof(int));      index\_proc = create\_shared\_memory(sizeof(int));      \*free\_index = 0;      \*index\_proc = 1;      shared\_memory = create\_shared\_memory(sizeof(int) \* THREADS\_AMOUNT);      shared\_memory\_proc = create\_shared\_memory(sizeof(int) \* OPERATIONS\_CHANGES);      for (int i = 0; i < THREADS\_AMOUNT; i++)      {          shared\_memory[i] = -1;      }      for (int i = 0; i < OPERATIONS\_CHANGES; i++)      {          shared\_memory\_proc[i] = -1;      }      pthread\_t threads[THREADS\_AMOUNT];      pthread\_attr\_t threads\_attrs[THREADS\_AMOUNT];      struct sched\_param params[THREADS\_AMOUNT];      int priorityFIFO = 50;      int priorityRR = 50;      int thread\_id[THREADS\_AMOUNT];      for (int i = 0; i < THREADS\_AMOUNT; i++)      {          pthread\_attr\_init(&threads\_attrs[i]);          pthread\_attr\_setinheritsched(&threads\_attrs[i], PTHREAD\_EXPLICIT\_SCHED);          params[i].sched\_priority = priorityFIFO;          pthread\_attr\_setschedpolicy(&threads\_attrs[i], SCHED\_FIFO);          pthread\_attr\_setschedparam(&threads\_attrs[i], &params[i]);          thread\_id[i] = i;          if(pthread\_create(&threads[i], &threads\_attrs[i], (i % 2 == 0) ? thread\_rr\_schedule : thread\_fifo\_schedule, &thread\_id[i]))          {              fprintf(stderr, "Error creating thread %d\n", i);          }      }      for (int i = 0; i < THREADS\_AMOUNT; i++)      {          pthread\_join(threads[i], NULL);      }      for (int i = 1; i < OPERATIONS\_CHANGES; i++)      {          if (shared\_memory\_proc[i] == -1)              break;          printf("%d ", shared\_memory\_proc[i]);      }      for (int i = 0; i < THREADS\_AMOUNT; i++)      {          printf("%s thread no %d complited\n", "FIFO", shared\_memory[i]);      }      printf("All threads finished\n");      return 0;  } |

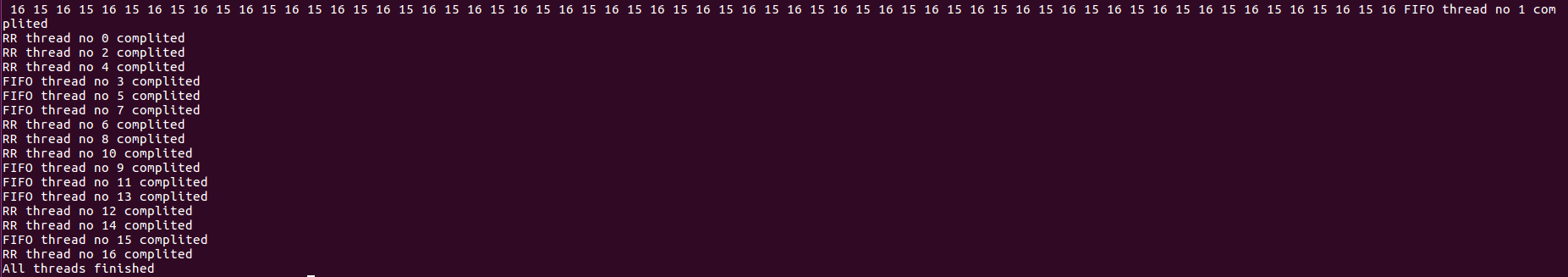




*rrfifo.c*

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <sched.h>  #include <errno.h>  #include <sys/mman.h>  #define THREADS\_AMOUNT 17  #define OPERATIONS\_AMOUNT 100000  #define OPERATIONS\_CHANGES THREADS\_AMOUNT\*OPERATIONS\_AMOUNT  int \*free\_index;  int \*shared\_memory;  int \*index\_proc;  int \*shared\_memory\_proc;  void\* create\_shared\_memory(size\_t size)  {      int protection = PROT\_READ | PROT\_WRITE;      int visibility = MAP\_SHARED | MAP\_ANONYMOUS;      return mmap(NULL, size, protection, visibility, -1, 0);  }  void \*thread\_fifo\_schedule(void \*attr)  {      for (int i = 0; i < OPERATIONS\_AMOUNT; i++)      {          if (shared\_memory\_proc[(\*index\_proc) - 1] != \*((int\*)attr))          {              shared\_memory\_proc[(\*index\_proc)++] = \*((int\*)attr);          }      }      shared\_memory[(\*free\_index)++] = \*((int\*)attr);  }  void \*thread\_rr\_schedule(void \*attr)  {      for (int i = 0; i < OPERATIONS\_AMOUNT; i++)      {          if (shared\_memory\_proc[(\*index\_proc) - 1] != \*((int\*)attr))          {              shared\_memory\_proc[(\*index\_proc)++] = \*((int\*)attr);          }      }      shared\_memory[(\*free\_index)++] = \*((int\*)attr);  }  int main()  {      free\_index = create\_shared\_memory(sizeof(int));      index\_proc = create\_shared\_memory(sizeof(int));      \*free\_index = 0;      \*index\_proc = 1;      shared\_memory = create\_shared\_memory(sizeof(int) \* THREADS\_AMOUNT);      shared\_memory\_proc = create\_shared\_memory(sizeof(int) \* OPERATIONS\_CHANGES);      for (int i = 0; i < THREADS\_AMOUNT; i++)      {          shared\_memory[i] = -1;      }      for (int i = 0; i < OPERATIONS\_CHANGES; i++)      {          shared\_memory\_proc[i] = -1;      }      pthread\_t threads[THREADS\_AMOUNT];      pthread\_attr\_t threads\_attrs[THREADS\_AMOUNT];      struct sched\_param params[THREADS\_AMOUNT];      int priorityFIFO = 50;      int priorityRR = 50;      int thread\_id[THREADS\_AMOUNT];      for (int i = 0; i < THREADS\_AMOUNT; i++)      {          pthread\_attr\_init(&threads\_attrs[i]);          pthread\_attr\_setinheritsched(&threads\_attrs[i], PTHREAD\_EXPLICIT\_SCHED);          if (i % 2 == 0)          {              params[i].sched\_priority = priorityRR;              pthread\_attr\_setschedpolicy(&threads\_attrs[i], SCHED\_RR);          }          else          {              params[i].sched\_priority = priorityFIFO;              pthread\_attr\_setschedpolicy(&threads\_attrs[i], SCHED\_FIFO);          }          pthread\_attr\_setschedparam(&threads\_attrs[i], &params[i]);          thread\_id[i] = i;          if(pthread\_create(&threads[i], &threads\_attrs[i], (i % 2 == 0) ? thread\_rr\_schedule : thread\_fifo\_schedule, &thread\_id[i]))          {              fprintf(stderr, "Error creating thread %d\n", i);          }      }      for (int i = 0; i < THREADS\_AMOUNT; i++)      {          pthread\_join(threads[i], NULL);      }      for (int i = 1; i < OPERATIONS\_CHANGES; i++)      {          if (shared\_memory\_proc[i] == -1)              break;          printf("%d ", shared\_memory\_proc[i]);      }      for (int i = 0; i < THREADS\_AMOUNT; i++)      {          printf("%s thread no %d complited\n", (shared\_memory[i] % 2 == 0) ? "RR" : "FIFO", shared\_memory[i]);      }      printf("All threads finished\n");      return 0;  } |

**

**

Наглядно продемонстрирована борьба за ресуср в реальном времени.

Представим борьбу за ресурсы без дополнительных вычислений.

*1.с*

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <sys/shm.h>  #include <sys/ipc.h>  #include <string.h>  #include <unistd.h>  int main()  {   int pid,pid2,pid3, n;   pid = fork();   if (pid == -1)   {     error("fork");     exit(1);   }  pid2 = fork();   if (pid2 == -1)   {     error("fork");     exit(1);   }   pid3 = fork();   if (pid3 == -1)   {     error("fork");     exit(1);   }  key\_t key = ftok("task1",65);  int shmid = shmget(key,1024,0666|IPC\_CREAT);  char \*str = (char\*) shmat(shmid,(void\*)0,0);    while(1){  //str = (char\*)"hello";  //printf("%s",str);      if (pid == 0) {               strcat(str,"0\n");        }      if(pid2 == 0){          strcat(str,"1\n");        }      if(pid3 == 0){          strcat(str,"2\n");        }     }     shmdt(str);   return 0;  } |

*out.c*

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <sys/shm.h>  #include <sys/ipc.h>  #include <string.h>  int main() {      // Создаем ключ и получаем идентификатор сегмента разделяемой памяти      key\_t key = ftok("task1", 65);      int shmid = shmget(key, 1024, 0666 | IPC\_CREAT);      // Получаем указатель на сегмент разделяемой памяти      char \*str = (char\*)shmat(shmid, NULL, 0);      // Выводим строку на экран      printf("Содержимое сегмента разделяемой памяти:\n%s", str);      // Отключаемся от сегмента разделяемой памяти      shmdt(str);      return 0;  } |





Теперь загрузим наши процессы разным количеством вычислений.

*2.c*

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <sys/shm.h>  #include <sys/ipc.h>  #include <string.h>  #include <unistd.h>  int main()  {   int pid,pid2,pid3, n;   pid = fork();   if (pid == -1)   {     error("fork");     exit(1);   }  pid2 = fork();   if (pid2 == -1)   {     error("fork");     exit(1);   }   pid3 = fork();   if (pid3 == -1)   {     error("fork");     exit(1);   }  key\_t key = ftok("task1",65);  int shmid = shmget(key,1024,0666|IPC\_CREAT);  char \*str = (char\*) shmat(shmid,(void\*)0,0);    while(1){      if (pid == 0) {              strcat(str,"0\n");          for(int i=0;i<100;i++){                  n\*=n;          }      }      if(pid2 == 0){              strcat(str,"1\n");              for(int i=0;i<1000;i++){                  n\*=n;          }      }      if(pid3 == 0){              strcat(str,"2\n");              for(int i=0;i<10000;i++){                  n\*=n;          }      }     }     shmdt(str);   return 0;  } |

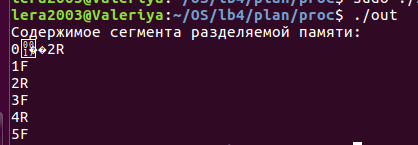


Как можно наблюдать, что при борьбе за ресурс приоритет отдаётся менее нагруженным процессам.

Теперь зададим обоим видам процессов равный приоритет.

3.c

|  |
| --- |
| #include <signal.h>  #include <pthread.h>  #include <stdio.h>  #include <sys/types.h>  #include <linux/unistd.h>  #include <sys/syscall.h>  #include <sched.h>  #include <errno.h>  #include <sys/shm.h>  #include <sys/ipc.h>  #include <unistd.h>  pthread\_t t[6];//1, t2,t3,t4;  char answer[10000];  void\* fifo(int t\_ind){      int k=3;      for(int i=0;i<1000000;i++){       k\*=k;      }      char c[3];           c[0] =    t\_ind+'0';      strcat(answer,c);        strcat(answer,"F\n");      pthread\_exit(NULL);  }  void\* rr(int t\_ind){      int k=3;      for(int i=0;i<1000000;i++){       k\*=k;      }        char c[3];      c[0] = t\_ind+'0';      strcat(answer,c);      strcat(answer,"R\n");      pthread\_exit(NULL);  }  void main()  {      key\_t key = ftok("task1",65);      int shmid = shmget(key,1024,0666|IPC\_CREAT);       char \*str = (char\*) shmat(shmid,(void\*)0,0);      int policy;      struct sched\_param param[6];      pthread\_attr\_t attr[6];//, attr\_2,attr\_3,;      int f = 30;      int r = 30;      for(int i=0;i<6;i++){       pthread\_attr\_init(&attr[i]);       pthread\_attr\_setinheritsched (&attr[i], PTHREAD\_EXPLICIT\_SCHED);       if(i%2 == 0){           pthread\_attr\_setschedpolicy(&attr[i], SCHED\_RR);           param[i].sched\_priority = r;           pthread\_attr\_setschedparam(&attr[i], &param[i]);           pthread\_create(&t[i], &attr[i], rr,i);       }else{           pthread\_attr\_setschedpolicy(&attr[i], SCHED\_FIFO);           param[i].sched\_priority = f;           pthread\_attr\_setschedparam(&attr[i], &param[i]);           pthread\_create(&t[i], &attr[i], fifo,i);       }  /\*      param[i].sched\_priority = r;    \*/      }      for(int i=0;i<6;i++){       pthread\_join(t[i],NULL);      }      strcpy(str,answer);      shmdt(str);      return 0;  } |

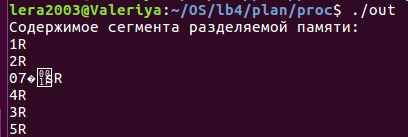


Как видно процессы работают в смешанном порядке.

Теперь оставим одну RR политику и сделаем большую нагрузку на процессы.

4.c

|  |
| --- |
| #include <signal.h>  #include <pthread.h>  #include <stdio.h>  #include <sys/types.h>  #include <linux/unistd.h>  #include <sys/syscall.h>  #include <sched.h>  #include <errno.h>  #include <sys/shm.h>  #include <sys/ipc.h>  #include <unistd.h>  pthread\_t t[6];//1, t2,t3,t4;  char answer[10000];  void\* fifo(int t\_ind){      int k=3;      for(int i=0;i<1000000;i++){       k\*=k;      }      char c[3];           c[0] =    t\_ind+'0';      strcat(answer,c);        strcat(answer,"F\n");      pthread\_exit(NULL);  }  void\* rr(int t\_ind){      int k=3;      for(int i=0;i<1000000;i++){       k\*=k;      }        char c[3];      c[0] = t\_ind+'0';      strcat(answer,c);      strcat(answer,"R\n");      pthread\_exit(NULL);  }  void main()  {      key\_t key = ftok("task1",65);      int shmid = shmget(key,1024,0666|IPC\_CREAT);       char \*str = (char\*) shmat(shmid,(void\*)0,0);      int policy;      struct sched\_param param[6];      pthread\_attr\_t attr[6];//, attr\_2,attr\_3,;      int f = 30;      int r = 30;      for(int i=0;i<6;i++){       pthread\_attr\_init(&attr[i]);       pthread\_attr\_setinheritsched (&attr[i], PTHREAD\_EXPLICIT\_SCHED);       if(1){           pthread\_attr\_setschedpolicy(&attr[i], SCHED\_RR);           param[i].sched\_priority = r;           pthread\_attr\_setschedparam(&attr[i], &param[i]);           pthread\_create(&t[i], &attr[i], rr,i);       }else{           pthread\_attr\_setschedpolicy(&attr[i], SCHED\_FIFO);           param[i].sched\_priority = f;           pthread\_attr\_setschedparam(&attr[i], &param[i]);           pthread\_create(&t[i], &attr[i], fifo,i);       }  /\*      param[i].sched\_priority = r;    \*/      }      for(int i=0;i<6;i++){       pthread\_join(t[i],NULL);      }      strcpy(str,answer);      shmdt(str);      return 0;  } |

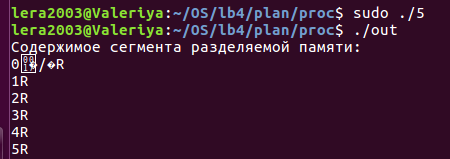


Можно заметить, что процессы выполнились в смешанном порядке за счёт перехода процессорного ресурса между ними, из-за большого количества вычислений.

Сделаем такой же эксперимент, но без нагрузки процессов.

5.c

|  |
| --- |
| #include <signal.h>  #include <pthread.h>  #include <stdio.h>  #include <sys/types.h>  #include <linux/unistd.h>  #include <sys/syscall.h>  #include <sched.h>  #include <errno.h>  #include <sys/shm.h>  #include <sys/ipc.h>  #include <unistd.h>  pthread\_t t[6];//1, t2,t3,t4;  char answer[10000];  void\* fifo(int t\_ind){      char c[3];      c[0] = t\_ind+'0';      strcat(answer,c);        strcat(answer,"F\n");      pthread\_exit(NULL);  }  void\* rr(int t\_ind){      char c[3];      c[0] = t\_ind+'0';      strcat(answer,c);      strcat(answer,"R\n");      pthread\_exit(NULL);  }  void main()  {      key\_t key = ftok("task1",65);      int shmid = shmget(key,1024,0666|IPC\_CREAT);       char \*str = (char\*) shmat(shmid,(void\*)0,0);      int policy;      struct sched\_param param[6];      pthread\_attr\_t attr[6];//, attr\_2,attr\_3,;      int f = 30;      int r = 30;      for(int i=0;i<6;i++){       pthread\_attr\_init(&attr[i]);       pthread\_attr\_setinheritsched (&attr[i], PTHREAD\_EXPLICIT\_SCHED);       if(1){           pthread\_attr\_setschedpolicy(&attr[i], SCHED\_RR);           param[i].sched\_priority = r;           pthread\_attr\_setschedparam(&attr[i], &param[i]);           pthread\_create(&t[i], &attr[i], rr,i);       }else{           pthread\_attr\_setschedpolicy(&attr[i], SCHED\_FIFO);           param[i].sched\_priority = f;           pthread\_attr\_setschedparam(&attr[i], &param[i]);           pthread\_create(&t[i], &attr[i], fifo,i);       }  /\*      param[i].sched\_priority = r;    \*/      }      for(int i=0;i<6;i++){       pthread\_join(t[i],NULL);      }      strcpy(str,answer);      shmdt(str);      return 0;  } |



Как видно, процессы выполнились в том же порядку, в каком начинались.