1. Разработать приложение, организующее неименованный канал между двумя процессами. Обмен данными должен выполняться в обе стороны (туда-обратно).
2. Разработать приложение, объединяющее 3 процесса в конвейер (родитель и два потомка, левый и правый), в котором поток стандартного вывода левого потомка соединен через канал с потоком стандартного ввода правого потомка. В левом и правом потомке д.б. предусмотрена возможность вызова (через exec) произвольных программ. Для демонстрации работы приложения реализовать выполнение в конвейере команды "ps ax | grep PID", где PID — идентификатор родительского процесса.
3. Разработать приложение, организующее именованный канал между двумя процессами, не связанными отношением «родитель-потомок». Обмен данными должен выполняться в обе стороны (туда-обратно).
4. Разработать приложение, организующее связь между двумя процессами с помощью очереди сообщений. Обмен данными должен выполняться в обе стороны (туда-обратно) с помощью разных типов сообщений (например, 10 – туда, 20 – обратно).
5. Разработать приложение, организующее синхронизацию между двумя процессами с помощью семафоров. Реализовать две ситуации: 1) проверка равенства семафора нулю и блокировкой одного из процессов; 2) попытки уменьшить значение семафора до отрицательного и последующей блокировки. Продемонстрировать успешную разблокировку процесса, выполнив печать сообщения в stdout.
6. Разработать приложение, организующее взаимодействие между двумя процессами с помощью разделяемой памяти. Обмен данными должен выполняться в обе стороны (туда-обратно).

# Приложение, организующее неименованный канал между двумя процессами

В рамках выполнения задания был изучен порядок создания неименованных каналов между процессами с помощью функции pipe() в среде CLion OC Linux. pipe - [системный вызов](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%BD%D1%8B%D0%B9_%D0%B2%D1%8B%D0%B7%D0%BE%D0%B2) в [Unix-подобных операционных системах](https://ru.wikipedia.org/wiki/Unix-%D0%BF%D0%BE%D0%B4%D0%BE%D0%B1%D0%BD%D0%B0%D1%8F_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), создающий неименованный канал,  который процесс использует для передачи сведений другим процессам.

Листинг 1. Приложение, организующее неименованный канал между двумя процессами

#include <unistd.h>

#include <sys/wait.h>

#include <stdio.h>

#include <stdlib.h>

int main() {

int pipe\_lr[2], pipe\_rl[2], status;

if (pipe(pipe\_lr) < 0) {

perror("error on pipe");

exit(1);

}

if (pipe(pipe\_rl) < 0) {

perror("error on pipe");

exit(1);

}

pid\_t pid\_l = fork();

if (pid\_l == -1) {

perror("error on fork");

return 1;

}

pid\_t pid\_r = fork();

if (pid\_r == -1) {

perror("error on fork");

return 1;

}

char buff\_l[1024];

char buff\_r[1024];

if (pid\_l == 0) {

close(pipe\_lr[0]);

close(pipe\_rl[1]);

char hello\_sec[] = "Hello, second!";

write(pipe\_lr[1], hello\_sec, sizeof(hello\_sec));

read(pipe\_rl[0], buff\_l, sizeof(buff\_l));

printf("1st read: %s\n", buff\_l);

exit(0);

} else if (pid\_r == 0) {

close(pipe\_lr[1]);

close(pipe\_rl[0]);

char hello\_first[] = "Hello, first!";

write(pipe\_rl[1], hello\_first, sizeof(hello\_first));

read(pipe\_lr[0], buff\_r, sizeof(buff\_r));

printf("2nd read: %s\n", buff\_r);

exit(0);

} else {

wait(&status);

return 0;

}

}

Результат выполнения листинга 1 приведен на рисунке 1.

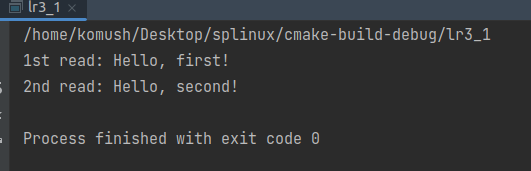


Рисунок 1. – Результат работы приложения

# Приложение, объединяющее 3 процесса в конвейер

В рамках разработки приложения был изучен порядок конвейерной работы процессов, в котором поток стандартного вывода левого потомка соединен через канал с потоком стандартного ввода правого потомка. В левом и правом потомке предусмотрена возможность вызова (через exec) произвольных программ. Для демонстрации работы приложения реализовать выполнение в конвейере команды "ps ax | grep PID", где PID — идентификатор родительского процесса. Семейство функций exec заменяет текущий образ процесса новым образом процесса. Новый образ должен быть создан из обычного исполняемого файла, называемого новым файлом образа процесса. Возврата от успешного выполнения не должно быть, поскольку образ вызывающего процесса накладывается на новый образ процесса.

Листинг 2. Приложение, организующее неименованный канал между двумя процессами.

#include <unistd.h>

#include <sys/wait.h>

#include <stdio.h>

#include <stdlib.h>

extern char \*\*environ;

int main() {

int pipe\_lr[2], status;

if (pipe(pipe\_lr) < 0) {

perror("error on pipe");

exit(1);

}

pid\_t pid\_l = fork();

if (pid\_l < 0) {

perror("error on fork");

return 1;

}

pid\_t pid\_r = fork();

if (pid\_r < 0) {

perror("error on fork");

return 1;

}

if (pid\_l == 0) {

close(pipe\_lr[0]);

close(1);

dup(pipe\_lr[1]);

close(pipe\_lr[1]);

char \*args[] = {"ps aux", 0};

if (execv("/bin/ps", args) < 0) {

perror("error on execve1");

exit(1);

}

exit(0);

}

if (pid\_r == 0) {

close(pipe\_lr[1]);

close(0);

dup(pipe\_lr[0]);

close(pipe\_lr[0]);

char ppid[5];

sprintf(ppid, "%d", getppid());

char \*args[] = {"grep", ppid, 0};

waitpid(pid\_l, &status, WUNTRACED | WCONTINUED);

if (execv("/bin/grep", args) < 0) {

perror("error on execve2");

exit(1);

}

exit(0);

}

wait(&status);

return 0;

}

Результат выполнения листинга 2 приведен на рисунке 2.

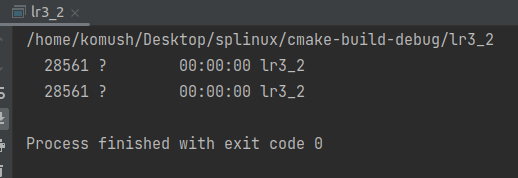


Рисунок 2. – Результат работы приложения

# Приложение, организующее именованный канал между двумя процессами, не связанными отношением «родитель-потомок».

В рамках выполнения задания был изучен порядок создания именованных каналов между процессами с помощью функции mkfifo() в среде CLion OC Linux. mkfifo - [системный вызов](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%BD%D1%8B%D0%B9_%D0%B2%D1%8B%D0%B7%D0%BE%D0%B2) в [Unix-подобных операционных системах](https://ru.wikipedia.org/wiki/Unix-%D0%BF%D0%BE%D0%B4%D0%BE%D0%B1%D0%BD%D0%B0%D1%8F_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), создающий именованный канал,  который процесс использует для передачи сведений другим процессам.

Листинг 3. Приложение, организующее именованный канал между двумя процессами

#include <unistd.h>

#include <sys/wait.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>

#include <fcntl.h>

int main() {

char buffer[24];

int status, fifofd, fifofd2;

if (mkfifo("myfifo1", 0777) < 0) {

perror("error on mkfifo1");

exit(1);

}

if (mkfifo("myfifo2", 0777) < 0) {

perror("error on mkfifo2");

exit(1);

}

pid\_t pid\_r;

pid\_t pid\_l = fork();

if (pid\_l < 0) {

perror("error on fork");

return 1;

}

if (pid\_l != 0){

pid\_r = fork();

if (pid\_r < 0) {

perror("error on fork");

return 1;

}

}

if (pid\_l == 0) {

fifofd = open("myfifo1", O\_WRONLY);

char hello[] = "Hello, second!";

write(fifofd, hello, sizeof(hello));

fifofd2 = open("myfifo2", O\_RDONLY);

read(fifofd2, buffer, sizeof(buffer));

printf("1st read: %s\n", buffer);

exit(0);

}

if (pid\_r == 0) {

fifofd = open("myfifo1", O\_RDONLY);

read(fifofd, buffer, sizeof(buffer));

printf("2nd read: %s\n", buffer);

fifofd = open("myfifo2", O\_WRONLY);

char hello[] = "Hello, first!";

write(fifofd, hello, sizeof(hello));

exit(0);

}

waitpid(pid\_l, &status, WUNTRACED | WCONTINUED);

waitpid(pid\_r, &status, WUNTRACED | WCONTINUED);

//wait(&status);

return 0;

}

Результат выполнения листинга 3 приведен на рисунке 3.

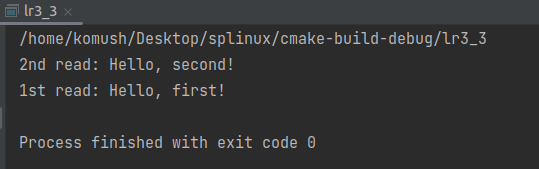


Рисунок 3. – Результат работы приложения

# Приложение, организующее связь между двумя процессами с помощью очереди сообщений

В рамках разработки приложения был изучен порядок создания очереди сообщений. Очереди сообщений представляют собой связный список в адресном пространстве ядра. Сообщения могут посылаться в очередь по порядку и доставаться из очереди несколькими разными путями. Каждая очередь сообщений однозначно определена идентификатором IPC.

msgget(key\_t key, int msgflags)- возвращает идентификатор очереди сообщений, связанный со значением параметра key . Она также создает новую очередь сообщений, если до этого такой не существовало.

int msgsnd(int msqid, const void \*msgp, size\_t msgsize, int flag)- системный вызов, добавляющий сообщение, указанное параметром msgp, в очередь сообщений, идентификатор которой указывается параметром msqid.

ssize\_t msgrcv(int msqid, void \*msgp, size\_t msgsize, long mtype, int flag) – системный вызов, добавляеющий сообщение, указанное параметром msgp, в очередь сообщений, идентификатор которой указывается параметром msqid.

Листинг 4. Приложение, организующее неименованный канал между двумя процессами

#include <sys/ipc.h>

#include <sys/msg.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/wait.h>

int main() {

int status;

pid\_t child = fork();

if (child < 0) {

perror("error on fork");

exit(1);

}

struct msgbuf {

long type;

char text[512];

} message;

if (child == 0) {

int msg\_child = msgget(1234, IPC\_CREAT | 0777);

if (msg\_child < 0) {

perror("error on msgget");

exit(1);

}

for (int i = 0; i < 20; ++i) {

strcpy(message.text, "hi, parent");

message.type = 10;

if (msgsnd(msg\_child, &message, sizeof(message.text), 0) < 0) {

perror("error on msgsnd");

exit(1);

}

}

for (int i = 0; i < 10; ++i) {

if (msgrcv(msg\_child, &message, sizeof(message.text), 11, 0) < 0) {

perror("error on msgrcv");

exit(1);

}

printf("message for child: %s\n", message.text);

}

exit(0);

} else {

int msg\_parent = msgget(1234, IPC\_CREAT | 0777);

if (msg\_parent < 0) {

perror("error on msgget");

exit(1);

}

for (int i = 0; i < 10; ++i) {

strcpy(message.text, "hi, child");

message.type = 11;

if (msgsnd(msg\_parent, &message, sizeof(message.text), 0) < 0) {

perror("error on msgsnd");

exit(1);

}

}

for (int i = 0; i < 20; ++i) {

if (msgrcv(msg\_parent, &message, sizeof(message.text), 10, 0) < 0) {

perror("error on msgrcv");

exit(1);

}

printf("message for parent: %s\n", message.text);

}

sleep(2);

}

return 0;

}

Результат выполнения листинга 4 приведен на рисунке 4.



Рисунок 4. – Результат работы приложения

# Приложение, организующее синхронизацию между двумя процессами с помощью семафоров

В рамках выполнения задания был изучен порядок работы с семафорами в среде CLion OC Linux. Семафор — примитив синхронизации работы [процессов](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) и [потоков](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%82%D0%BE%D0%BA_%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F), в основе которого лежит счётчик, над которым можно производить две [атомарные операции](https://ru.wikipedia.org/wiki/%D0%90%D1%82%D0%BE%D0%BC%D0%B0%D1%80%D0%BD%D0%B0%D1%8F_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D1%8F): увеличение и уменьшение значения на единицу, при этом операция уменьшения для нулевого значения счётчика является блокирующейся. Служит для построения более сложных механизмов синхронизации и используется для синхронизации параллельно работающих задач, для защиты передачи данных через [разделяемую память](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D1%8F%D0%B5%D0%BC%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C), для защиты [критических секций](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%81%D0%B5%D0%BA%D1%86%D0%B8%D1%8F), а также для управления доступом к аппаратному обеспечению.

int semget(key\_t key, int nsems, int semflag)- системный вызов возвращает идентификатор набора семафоров, связанный с аргументом key. Создается новый набор, если такого не существует.

int semop(int semid, struct sembuf \*sops, unsigned nsops)- функция производит операции над выбранными элементами из набора семафоров semid. Каждый из элементов nsops в массиве sops определяет операцию, производимую над семафором в структуре struct sembuf, состоящей из полей:

short sem\_num;   /\* semaphore number: 0 = first \*/  
short sem\_op;    /\* semaphore operation \*/  
short sem\_flg;   /\* operation flags \*/

Листинг 5. Приложение, организующее связь между двумя процессами с помощью очереди сообщений

#include <sys/ipc.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/sem.h>

#include <sys/wait.h>

#include <string.h>

int main() {

int status;

pid\_t pid\_l = fork();

if (pid\_l < 0) {

perror("error on fork");

exit(1);

}

pid\_t pid\_r = fork();

if (pid\_r < 0) {

perror("error on fork");

exit(1);

}

if (pid\_l == 0) {

// уменьшить значение 1-го семафора на 1

// проверить значение 2-го семафора на равенство нулю

struct sembuf sbuf[2] = {{0, -1, SEM\_UNDO},

{1, 0, 1}};

// создается набор из двух семафоров

int semid = semget(100, 2, IPC\_CREAT | 0777);

if (semid < 0) {

perror("error on semget");

exit(1);

}

if (semop(semid, sbuf, 2) < 0) {

perror("error on semop");

exit(1);

}

// в этом месте процесс X будет заблокирован

printf("I was unlocked");

exit(0);

} else if (pid\_r == 0) {

// увеличить значение семафора на 1

struct sembuf sbuf[1] = {0, 1, SEM\_UNDO};

int semid = semget(100, 2, 0777);

if (semid < 0) {

perror("error on semget");

exit(1);

}

printf("My pid is %d, and i will unlock %d", getpid(), pid\_l);

sleep(5);

if (semop(semid, sbuf, 1) < 0) {

perror("error on semop");

exit(1);

}

// в этом месте процесс X разблокируется

exit(0);

} else {

wait(&status);

// waitpid(pid\_r, &status, WUNTRACED | WCONTINUED);

// waitpid(pid\_l, &status, WUNTRACED | WCONTINUED);

return 0;

}

}

Результат выполнения листинга 5 приведен на рисунке 5.

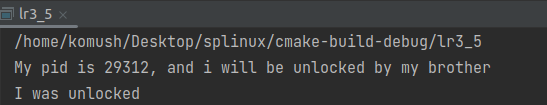


Рисунок 5. – Результат работы приложения

# Приложение, организующее взаимодействие между двумя процессами с помощью разделяемой памяти

В рамках выполнения задания был изучен порядок работы с разделяемой памятью в среде CLion OC Linux. Техника разделяемой памяти позволяет осуществлять обмен информацией через общий для процессов сегмент памяти без использования системных вызовов ядра. Сегмент разделяемой памяти подключается в свободную часть [виртуального адресного пространства](https://ru.wikipedia.org/wiki/%D0%90%D0%B4%D1%80%D0%B5%D1%81%D0%B0%D1%86%D0%B8%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D0%B8#%D0%B2%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F) процесса. Таким образом, два разных процесса могут иметь разные адреса одной и той же ячейки подключенной разделяемой памяти.

int shmget(key\_t key, int size, int shmflag) - возвращает идентификатор разделяемому сегменту памяти, соответствующий значению аргумента key. Создает новый, если такого не существовало.

void\* shmat(int shmid, void \*address, int flag) - подстыковывает сегмент разделяемой памяти shmid к адресному пространству вызывающего процесса.

Листинг 6. Приложение, организующее взаимодействие между двумя процессами с помощью разделяемой памяти

#include <sys/ipc.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/sem.h>

#include <sys/wait.h>

#include <sys/shm.h>

#include <string.h>

int main() {

int status;

pid\_t pid\_l = fork();

if (pid\_l < 0) {

perror("error on fork");

exit(1);

}

pid\_t pid\_r;

if (pid\_l != 0) {

pid\_r = fork();

if (pid\_r < 0) {

perror("error on fork");

exit(1);

}

}

if (pid\_l == 0) {

char \*shmem;

// создать область разделяемой памяти размеров 1 Кб

int shmid = shmget(102, 1024, IPC\_CREAT | 0777);

if (shmid < 0) {

perror("error on shmget");

exit(1);

}

// подсоединяемся к разделяемой памяти

if ((shmem = (char \*) shmat(shmid, 0, 0)) < 0) {

perror("error on shmat");

exit(1);

}

// пишем в разделяемую память

printf("String from shared memory: %s\n", shmem);

strcpy(shmem, "Hello, second!");

shmdt((void \*) shmem);

exit(0);

}

if (pid\_r == 0) {

char \*shmem;

int shmid = shmget(102, 1024, 0777);

if (shmid < 0) {

perror("error on shmget");

exit(1);

}

// подсоединяемся к разделяемой памяти

if ((shmem = (char \*) shmat(shmid, 0, 0)) < 0) {

perror("error on shmat");

exit(1);

}

// читаем разделяемую память

printf("String from shared memory2: %s\n", shmem);

strcpy(shmem, "Hello, first!");

shmdt((void \*) shmem);

exit(0);

}

wait(&status);

return 0;

}

Результат выполнения листинга 6 приведен на рисунке 6.

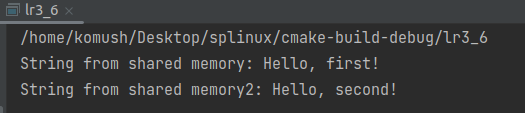


Рисунок 6. – Результат работы приложения

# Вывод

Было разработано приложение, организующее неименованный канал между двумя процессами. Обмен данными выполняется в обе стороны (туда-обратно).

Так же разработано приложение, объединяющее 3 процесса в конвейер (родитель и два потомка, левый и правый), в котором поток стандартного вывода левого потомка соединен через канал с потоком стандартного ввода правого потомка. В левом и правом потомке предусмотрена возможность вызова (через exec) произвольных программ. Для демонстрации работы приложения реализовано выполнение в конвейере команды "ps ax | grep PID", где PID — идентификатор родительского процесса.

Помимо этого, создано приложение, организующее именованный канал между двумя процессами, не связанными отношением «родитель-потомок». Обмен данными выполняться так же в обе стороны.

Разработано приложение, организующее связь между двумя процессами с помощью очереди сообщений. Обмен данными должен выполняться в обе стороны (туда-обратно) с помощью разных типов сообщений (например, 10 – туда, 20 – обратно).

Разработано приложение, организующее синхронизацию между двумя процессами с помощью семафоров. Рассмотрены две ситуации: 1) проверка равенства семафора нулю и блокировкой одного из процессов; 2) попытки уменьшить значение семафора до отрицательного и последующей блокировки.

Разработано приложение, организующее взаимодействие между двумя процессами с помощью разделяемой памяти. Обмен данными выполняется в обе стороны.