МИНИСТЕРСТВО НАУКИ
И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

НГТУ
НЭТИ | **Факультет прикладной математики и информатики**

Кафедра прикладной математики

Лабораторная работа № 3

по дисциплине «Управление ресурсами в вычислительных системах»

СИНХРОНИЗАЦИЯ ПРОЦЕССОВ

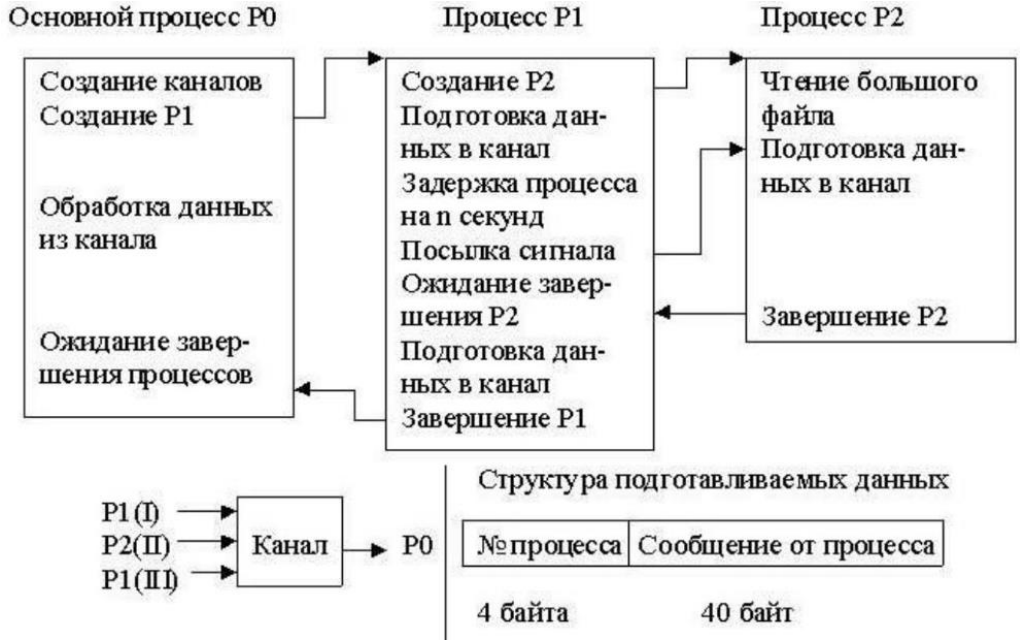
Бригада 7	ГРУШЕВ АНДРЕЙ
Группа ПМ-05	БОЛДЫРЕВ СЕРГЕЙ
Вариант 7	ХАБАРОВА АНАСТАСИЯ

Преподаватели	СТАСЫШИН ВЛАДИМИР МИХАЙЛОВИЧ
	СИВАК МАРИЯ АЛЕКСЕЕВНА

Новосибирск, 2023

Условие

Исходный процесс создает программный канал К1 и порождает новый процесс Р1, а тот, в свою очередь, порождает ещё один процесс Р2. Подготовленные данные последовательно помещаются процессами-сыновьями в программный канал и передаются основному процессу. Файл, читаемый процессом Р2, должен быть достаточно велик с тем, чтобы его чтение не завершилось ранее, чем закончится установленная задержка в n секунд. После срабатывания будильника процесс Р1 посылает сигнал процессу Р2, прерывая чтение файла. Схема взаимодействия процессов, порядок передачи данных в канал и структура подготавливаемых данных показаны ниже:



Обработка данных основным процессом заключается в чтении информации из программного канала и печати её. Кроме того, посредством выдачи сообщений необходимо информировать обо всех этапах работы программы (создание процесса, завершение посылки данных в канал и т.д.).

Используемые программные средства

`signal(int sig, func)` – системный вызов, позволяющий процессу самостоятельно определить свою реакцию на получение сигнала. Реакцией процесса, осуществившего системный вызов с аргументом `func`, при получении сигнала `sig` будет вызов функции `func()`.

`pipe(fd)` – системный вызов, возвращающий два дескриптора файла: для записи данных в канал и для чтения.

`fork(void)` – системный вызов, порождающий новый дочерний процесс.

`wait(int *status)` – системный вызов, с помощью которого выполняется ожидание завершения процесса-потомка родительским процессом.

`exit(int status)` – системный вызов, предназначенный для завершения функционирования процесса. Аргумент `status` является статусом завершения, который передается родительскому процессу, если он выполнял системный вызов `wait`.

`fprintf(FILE* stream, const char *format, …)` – форматированный вывод в файл на который указывает `stream`.

`printf(const char* format, …)` – форматированный вывод в файл стандартного вывода.

`sprintf(char* str, const char* format, …)` – форматированный вывод в символьную строку на которую указывает `str`.

read(int fd, void *buf, size_t count) – производит запись count байтов файлового описателя fd в буфер, адрес которого начинается с buf.

write(int fd, const void *buf, size_t count) – производит запись в описатель файла. Возвращает количество записанных байтов в случае успешного завершения, иначе возвращает -1.

getpid(void) – возвращает идентификатор текущего процесса.

alarm(n) – системный вызов, обеспечивающий посылку процессу сигнала SIGALARM через n секунд.

pause() – системный вызов, позволяющий приостановить процесс до тех пор, пока не будет получен какой-либо сигнал.

kill(int pid, int sig) – системный вызов, посылающий сигнал, специфицированный аргументом sig, процессу, который имеет идентификатор pid или группе процессов.

open(const char *pathname, int flags) – преобразовывает путь к файлу в описатель файла. Возвращает файловый описатель, который не открыт процессом. Функция с флагом O_TRUNC урезает длину файла до нуля, если файл существует, является обычным файлом и режим позволяет запись в этот файл.

Алгоритм решения

1. Исходный процесс создает программный канал и порождает новый процесс.
2. Дочерний процесс порождает новый процесс, помещает подготовленные данные в канал.
3. Внучатый процесс помещает подготовленные данные в канал и завершается.
4. Дочерний процесс возобновляется после получения сигнала, помещает подготовленные данные в канал и завершается.
5. Родительский процесс при каждом обновлении содержимого программного канала считывает из него информацию и выводит на экран.

Спецификация

Код программы расположен на сервере НГТУ в директории **/home/NSTU/pmi-b0507/upres/lab3**.
Файлы с кодом на языке **C** – **main.c**.

Для корректной работы программы необходимо, чтобы в папке с исполняемым файлом находился файл большой размерности с наименованием **big_file**. Для заполнения файла большой размерности существует вспомогательная программа **recorder.c**. Для получения исполняемого файла необходимо, находясь в директории с файлом **recorder.c,** выполнить команду:
gcc recorder.c -o [имя_исполняемого_файла],
либо воспользоваться make-файлом при помощи команды:
make recorder,
которая создаст исполняемый файл **recorder.o**.

Для получения основного исполняемого файла необходимо, находясь в директории с файлом **main.c,** выполнить команду:
gcc main.c -o [имя_исполняемого_файла],
либо воспользоваться make-файлом при помощи команды:
make main,
которая создаст исполняемый файл **main.o**.

Также можно воспользоваться командой **make all,**
которая создаст исполняемые файлы **main.o** и **recorder.o**.

Запуск программы происходит при помощи команды:
./[имя_исполняемого_файла],
например:
./main.o

Формат вывода результата:

[идентификатор процесса]: [сообщение от процесса]
…

Тестирование программы:

№	Входные данные	Результаты работы программы
1	./main.o	P1 sucessfully created P2 sucessfully created P1 successfully wrote own pid to pipe P1 successfully wrote own message to pipe P1 turn on an alarm for 3 seconds P0 successfully read a pid from pipe P0 successfully read a message from pipe 6117: (P1) Process is created P2 successfully opened big file P2 started reading file P1 is awake P1 successfully sent SIGINT to P2 P2 finished reading file P2 successfully wrote own pid to pipe P2 successfully wrote own message to pipe P0 successfully read a pid from pipe P0 successfully read a message from pipe 6120: (P2) Process read 1637364 times P1 successfully wrote own pid to pipe P1 successfully wrote own message to pipe P0 successfully read a pid from pipe P0 successfully read a message from pipe 6117: (P1) Process 6120 ended with status 0

Make-файлы

Файл **makefile:**

```
main: main.c
    gcc main.c -o main.o

recorder: recorder.c
    gcc recorder.c -o recorder.o

all: main.c recorder.c
    gcc main.c -o main.o
    gcc recorder.c -o recorder.o

clean:
    rm main.o
    rm recorder.o
```

Листинг программы

main.c:

```c
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <signal.h>
#include <stdbool.h>

bool doReading = true; //Переменная-флаг для чтения файла

/**
 * Функция-обработчик сигнала SIGLARM
**/
void sigALRMHandler()
{
    signal(SIGALRM, sigALRMHandler);
}

/**
 * Функция-обработчик сигнала SIGINT
**/
void sigINTHandler()
{
    doReading = false; //Останавливаем чтение файла
    signal(SIGINT, sigINTHandler);
}

int main(int argc, char** argv)
{
    int k1[2];
    pipe(k1); //Канал для передачи сообщений между процессами

    signal(SIGALRM, sigALRMHandler); //Назначение обработчика сигнала SIGALRM
    signal(SIGINT, sigINTHandler); //Назначение обработчика сигнала SIGINT

    pid_t p1 = fork();

    if (-1 == p1)
    {
        fprintf(stderr, "P0: Unable to fork P1.\n");
        exit(EXIT_FAILURE);
    }
    else if (p1 > 0)    //Процесс P0
    {
        printf("P1 sucessfully created\n");

        char recievedMsg[40]; //Буфер для полученного сообщение
        int i, status, spid; //spid - sender process id
```

```
        for (i = 0; i < 3; i++) //Планируется получить 3 сообщения
        {
            if (read(k1[0], &spid, sizeof(int)) == sizeof(int))
                printf("P0 successfully read a pid from pipe\n");
            else
                fprintf(stderr, "P0 cannot read a pid from pipe\n");

            if (read(k1[0], &recievedMsg, 40) == 40)
                printf("P0 successfully read a message from pipe\n");
            else
                fprintf(stderr, "P0 cannot read a message from pipe\n");

            printf("%d: %s\n", spid, recievedMsg);
        }

        wait(&status); //Ожидание завершения процесса P1

        if (status == EXIT_FAILURE)
            exit(EXIT_FAILURE);

        exit(EXIT_SUCCESS);
    }
    else    //Процесс P1
    {
        pid_t p2 = fork();
        if (-1 == p2)
        {
            fprintf(stderr, "P1: Unable to fork P2.\b");
            exit(EXIT_FAILURE);
        }
        else if (p2 > 0) // Процесс P1
        {
            printf("P2 sucessfully created\n");
            char msg[40];
            pid_t pid = getpid();
            int status;

            sprintf(msg, "(P1) Process is created\0");

            if (write(k1[1], &pid, sizeof(int)) == sizeof(int)) //Отправка сообщения
через канал
                printf("P1 successfully wrote own pid to pipe\n");
            else
                fprintf(stderr, "P1 cannot write own pid to pipe\n");

            if (write(k1[1], &msg, 40) == 40)
                printf("P1 successfully wrote own message to pipe\n");
            else
                fprintf(stderr, "P1 cannot write own message to pipe\n");
```

```
            alarm(3); //Будильник на 3 секунды
            printf("P1 turn on an alarm for 3 seconds\n");
            pause();
            printf("P1 is awake\n");

            if (-1 == kill(p2, SIGINT)) //Отправка SIGINT дочернему процессу
                fprintf(stderr, "%s\n", strerror(errno));
            else
                printf("P1 successfully sent SIGINT to P2\n");

            wait(&status); //Ожидание завершения дочернего процесса

            sprintf(msg, "(P1) Process %d ended with status %d\0", p2, status);

            if (write(k1[1], &pid, sizeof(int)) == sizeof(int)) //Отправка сообщения
через канал
                printf("P1 successfully wrote own pid to pipe\n");
            else
                fprintf(stderr, "P1 cannot write own pid to pipe\n");

            if (write(k1[1], &msg, 40) == 40)
                printf("P1 successfully wrote own message to pipe\n");
            else
                fprintf(stderr, "P1 cannot write own message to pipe\n");

            exit(EXIT_SUCCESS);
        }
        else    //  Процесс P2
        {
            int bf = open("big_file", O_RDONLY); //Открытие файла большого размера

            if (-1 == bf)
            {
                fprintf(stderr, "P2 cannot open big file\n");
                exit(EXIT_FAILURE);
            }
            else
                printf("P2 successfully opened big file\n");

            int number, i = 0;
            char msg[40];
            pid_t pid = getpid();

            printf("P2 started reading file\n");

            while(doReading)
                if (read(bf, &number, sizeof(int)) != sizeof(int)) //Чтение из файла,
пока процесс не получит сигнал
                    doReading = false;
                else
                    i++;
```

```
            printf("P2 finished reading file\n");

            sprintf(msg, "(P2) Process read %d times\0", i);

            if (write(k1[1], &pid, sizeof(int)) == sizeof(int)) //Отправка сообщения
через канал
                printf("P2 successfully wrote own pid to pipe\n");
            else
                fprintf(stderr, "P2 cannot write own pid to pipe\n");

            if (write(k1[1], &msg, 40) == 40)
                printf("P2 successfully wrote own message to pipe\n");
            else
                fprintf(stderr, "P2 cannot write own message to pipe\n");

            exit(EXIT_SUCCESS);
        }
    }
    return 0;
}
```

recorder.c:

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>

int main(int argc, char** argv)
{
    int result = 0;
    int fd = open("big_file", O_RDONLY);
    while (read(fd, &result, sizeof(int)) == sizeof(int))
    {
        ;
    }
    close(fd);
    return 0;
}
```