ЛР4

Задание 1

Необходимые знания

1. Функция `kill`
2. Неблокирующий wait с `WNOHANG`
3. Функция `alarm`, сигнал `SIGALRM`, функция `signal`.

Дополнить программу parallel_min_max.c из ***лабораторной работы №3***, так чтобы после заданного таймаута родительский процесс посылал дочерним сигнал SIGKILL. Таймаут должен быть задан, как именной необязательный параметр командной строки (`--timeout 10`). Если таймаут не задан, то выполнение программы не должно меняться.

1. Функция kill()

Назначение:
Отправка сигналов процессам.

- **pid**: ID процесса (>0 — конкретный процесс, 0 — всем процессам группы, -1 — всем доступным процессам).
- **sig**: Номер сигнала (например, SIGTERM, SIGKILL).

2. Неблокирующий wait с WNOHANG

Назначение: Проверка статуса дочерних процессов без блокировки.

3. Функция alarm(), сигнал SIGALRM, signal()

alarm()

Назначение:
Установка таймера для отправки сигнала SIGALRM через указанное время.

Сигнал SIGALRM

Стандартный сигнал для уведомления о срабатывании таймера.

Функция signal()

Назначение:
Установка обработчика сигнала.

Программу написал

Задание 2

Необходимые знания

1. Что такое зомби процессы, как появляются, как исчезают.

Создать программу, с помощью которой можно продемонстрировать зомби процессы. Необходимо объяснить, как появляются зомби процессы, чем они опасны, и как можно от них избавиться.

Зомби-процессы (Zombie Processes)

1. Что это?

Зомби-процесс — это **уже завершенный процесс**, который остается в таблице процессов до тех пор, пока родительский процесс не прочитает его статус завершения.

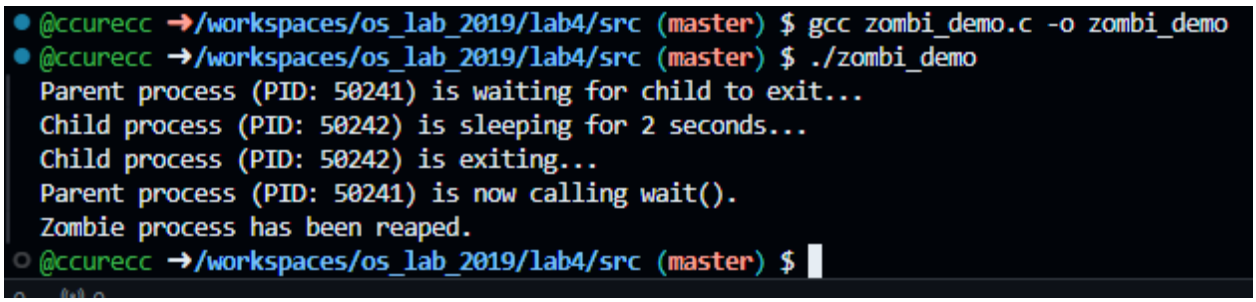
2. Как появляются?

1. Дочерний процесс завершается (вызовом exit() или сигналом).
2. Родительский процесс **не вызывает** wait() или waitpid() для чтения статуса.
3. Ядро сохраняет запись о процессе (PID, код завершения), пока родитель не обработает его.

3. Как исчезают?

- Родитель вызывает wait()/waitpid() — зомби удаляется из таблицы процессов.

- Родитель завершается — все зомби-потомки переходят к init (PID 1), который автоматически вызывает wait().



Задание 3

Необходимые знания

1. Работа виртуальной памяти.

Скомпилировать process_memory.c. Объяснить, за что отвечают переменные `etext`, `edata`, `end`.

1. Что такое виртуальная память?

Это абстракция, предоставляемая ОС, которая позволяет программам "думать", что у них есть **непрерывное адресное пространство**, даже если физическая память фрагментирована или занята другими процессами.

1. etext - адрес конца сегмента кода (текстового сегмента). Это точка, где заканчивается исполняемый код программы.
2. edata - адрес конца сегмента инициализированных данных. Это граница между инициализированными и неинициализированными данными.
3. end - адрес конца сегмента BSS (неинициализированных данных) и начала кучи (heap).

Задание 4

Создать makefile, который собирает программы из задания 1 и 3.

Задание 6

Создать makefile для parallel_sum.c.

Выполнил эти задания в одном файле

```
# Компилятор и флаги
CC = gcc
CFLAGS = -I. -Wall -Wextra -pthread

# Цели
all: parallel_min_max process_memory parallel_sum

# Сборка программы parallel_min_max
parallel_min_max: parallel_min_max.o find_min_max.o utils.o
    $(CC) -o parallel_min_max parallel_min_max.o find_min_max.o utils.o $(CFLAGS)

# Сборка программы process_memory
process_memory: process_memory.o
    $(CC) -o process_memory process_memory.o $(CFLAGS)

# Сборка программы parallel_sum
parallel_sum: parallel_sum.o
    $(CC) -o parallel_sum parallel_sum.o $(CFLAGS)

# Правила для сборки объектов
parallel_min_max.o: parallel_min_max.c find_min_max.h utils.h
    $(CC) -c parallel_min_max.c $(CFLAGS)

find_min_max.o: find_min_max.c find_min_max.h utils.h
    $(CC) -c find_min_max.c $(CFLAGS)

utils.o: utils.c utils.h
    $(CC) -c utils.c $(CFLAGS)

process_memory.o: process_memory.c
    $(CC) -c process_memory.c $(CFLAGS)

parallel_sum.o: parallel_sum.c
    $(CC) -c parallel_sum.c $(CFLAGS)

# Очистка
clean:
    rm -f *.o parallel_min_max process_memory parallel_sum
```

Задание 5

Необходимые знания

1. POSIX threads: как создавать, как дожидаться завершения.
2. Как линковаться на бибилотеку `pthread`

Доработать parallel_sum.c так, чтобы:

- Сумма массива высчитывалась параллельно.
- Массив генерировался с помощью функции `GenerateArray` из ***лабораторной работы №3***.
- Программа должна принимать входные аргументы: количество потоков, seed для генерирования массива, размер массива (`./psum --threads_num "num" --seed "num" --array_size "num"`).
- Вместе с ответом программа должна выводить время подсчета суммы (генерация массива не должна попадать в замер времени).
- Вынести функцию, которая считает сумму в отдельную библиотеку.

Работа с POSIX Threads (pthreads)

1. Создание потока

Используется функция pthread_create().

```
#include <pthread.h>

int pthread_create(
    pthread_t *           ,          // Указатель на идентификатор потока
    const pthread_attr_t *      , // Атрибуты потока (NULL по умолчанию)
    void *(*                    )(void *), // Функция, которую выполнит поток
    void *                      // Аргумент для функции потока
);
```

2. Ожидание завершения потока

Функция pthread_join() блокирует выполнение, пока поток не завершится.

```
int pthread_join(
    pthread_t           ,   // Идентификатор потока
    void **               // Указатель на возвращаемое значение (можно NULL)
);
```

3. Как линковаться с библиотекой pthread

При компиляции нужно добавить флаг -pthread (или -lpthread в старых версиях GCC).

gcc program.c –o program -pthread

Сделал