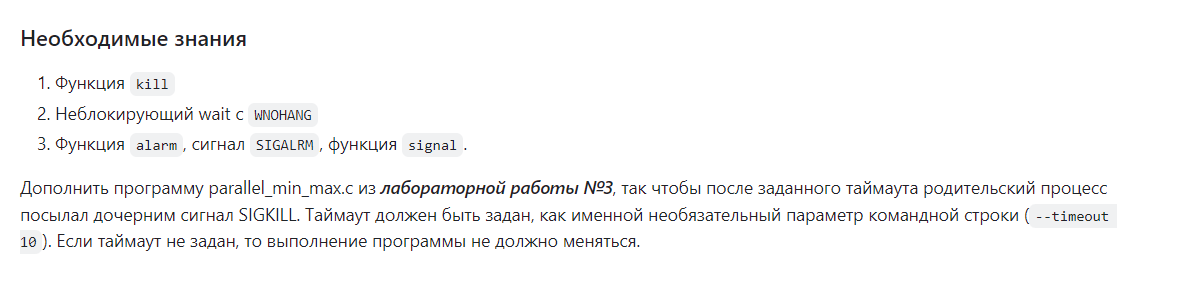
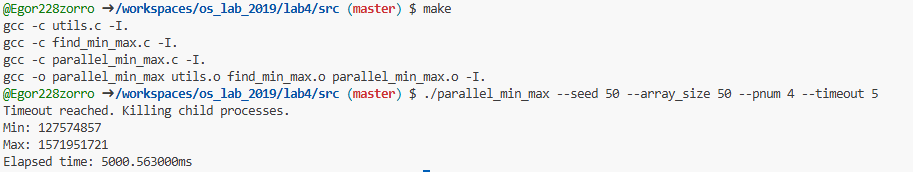
**Лабораторная работа №4**

**Заданиe 1**

****



**1.1 Функция kill**

**kill — это системный вызов, который используется для отправки сигналов процессам или группам процессов. Функция не обязательно завершает процесс, а просто отправляет ему сигнал. Конечное поведение зависит от типа сигнала и того, как процесс его обрабатывает.**

**pid — ID процесса (>0 — конкретный процесс;0 — всем процессам группы; -1 — всем доступным процессам).**

**sig — номер сигнала, который будет отправлен (например, SIGKILL, SIGTERM, SIGSTOP)**

**SIGKILL — сигнал, который мы отправляем процессу. В данном случае это сигнал немедленного завершения.**

**Примеры сигналов:**

**SIGKILL — немедленное завершение процесса.**

**SIGTERM — запрос на завершение процесса.**

**SIGSTOP — приостановка выполнения процесса.**

**SIGALRM — сигнал от таймера (например, используется в вашей программе с alarm()).**

**1.2 Неблокирующий wait с WNOHANG**

**Системный вызов wait используется для ожидания завершения дочерних процессов. По умолчанию он блокирует выполнение родительского процесса до завершения одного из его дочерних процессов. Однако с использованием флага WNOHANG можно сделать вызов wait неблокирующим.(** **Проверка статуса дочерних процессов без блокировки)**

**Флаг WNOHANG**

**WNOHANG сообщает системе, что если нет завершённых дочерних процессов, wait должен немедленно вернуть управление родительскому процессу, а не блокироваться в ожидании.**

**Это удобно, когда родительский процесс должен выполнять другие задачи, не ожидая завершения дочерних процессов.**

**Различия между блокирующим и неблокирующим wait:**

| **Блокирующий wait** | **Неблокирующий waitpid с WNOHANG** |
| --- | --- |
| **Родительский процесс останавливается, пока не завершится хотя бы один дочерний процесс.** | **Родитель продолжает выполнение, даже если нет завершившихся дочерних процессов.** |
| **Просто реализуется.** | **Требует проверки результата и обработки случаев, когда дочерние процессы ещё не завершены.** |
| **Подходит, если родительский процесс ничего больше не делает.** | **Удобно для выполнения других задач параллельно с ожиданием.** |

**1.3 Функция alarm, сигнал SIGALRM, функция signal**

**Функция alarm**

**Функция alarm устанавливает таймер .По истечении указанного времени отправляется сигнал SIGALRM процессу, который вызвал alarm.**

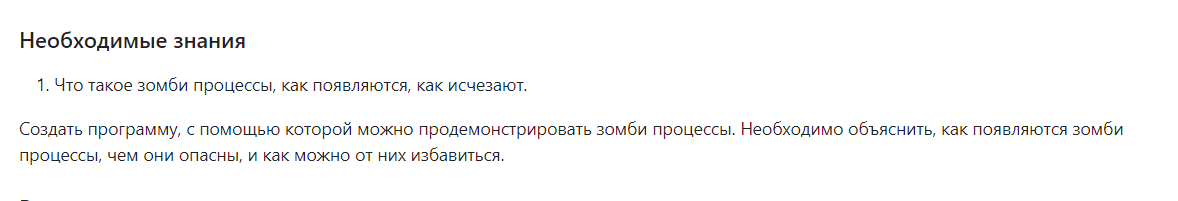
**Сигнал SIGALRM**

**Сигнал SIGALRM уведомляет процесс о том, что истёк таймер, установленный функцией alarm.(** **Стандартный сигнал для уведомления о срабатывании таймера)**

**Функция signal()**

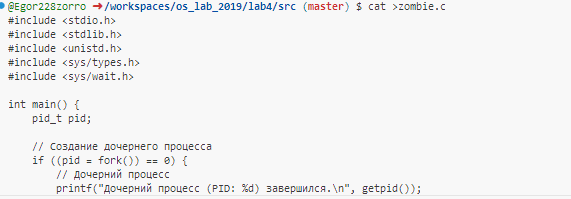
**Устанавливает обработчик сигнала.**

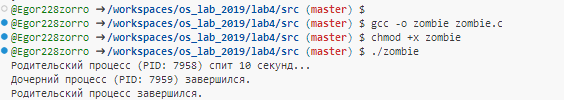
**Заданиe 2**

****

**Что такое зомби-процессы?**

**Зомби-процесс— это процесс, который завершился, но его запись в таблице процессов остаётся до тех пор, пока родительский процесс не вызовет wait() или waitpid() для получения его статуса завершения.**

****

****

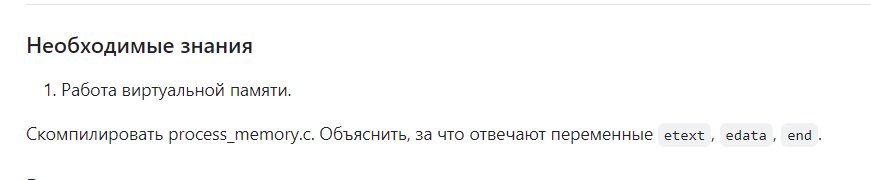
**Как появляются зомби-процессы?**

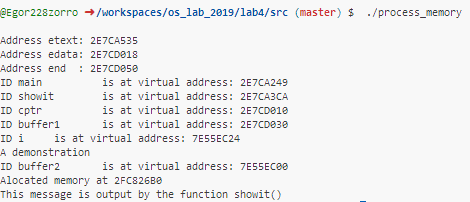
Дочерний процесс завершает свою работу с помощью exit(0). Он становится зомби, так как родительский процесс не вызвал wait() или waitpid() для чтения статуса. Ядро сохраняет запись о процессе (PID, код завершения), пока родитель не обработает его.

**Чем опасны?** Каждый зомби-процесс занимает запись в таблице процессов. Если зомби-процессов становится слишком много, это может привести к исчерпанию ресурсов и невозможности создания новых процессов.

**Как избавиться**? Родительский процесс должен вызывать wait() или waitpid(), чтобы получить статус завершения дочернего процесса(зомби удаляется из таблицы процессов). Родитель завершается — все зомби-потомки переходят к init (PID 1), который автоматически вызывает wait().

**Заданиe 3**

****

****

**В process\_memory.c** выводятся адреса различных переменных и функции в памяти процесса.

**etext:** Указывает на адрес конца сегмента текста (код программы). Это область памяти, где хранится исполняемый код.

**edata:** Указывает на адрес конца сегмент инициализированных данных. Это область памяти, где хранятся глобальные и статические переменные, которые были инициализированы. (Это граница между инициализированными и неинициализированными данными)

**end:** Указывает на адрес конца сегмента неинициализированных данных (bss) и начала кучи (heep). Это область памяти, где хранятся глобальные и статические переменные, которые не были **инициализированы.**

**1. Работа виртуальной памяти**

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

**Заданиe 4**

****

**CC=gcc**

**CFLAGS=-I.**

**new : parallel\_min\_max process\_memory zombie**

**@echo "Запуск parallel\_min\_max с параметрами: $(ARGS)"**

**@./parallel\_min\_max $(ARGS)**

**@echo "Запуск process\_memory..."**

**@sleep 3**

**@./process\_memory**

**@echo "Запуск zombie..."**

**@sleep 3**

**@./zombie**

**parallel\_min\_max : parallel\_min\_max.c utils.o find\_min\_max.o utils.h find\_min\_max.h**

**$(CC) -o parallel\_min\_max utils.o find\_min\_max.o parallel\_min\_max.c $(CFLAGS)**

**zombie : zombie.c**

**$(CC) -o zombie zombie.c $(CFLAGS)**

**process\_memory : process\_memory.c**

**$(CC) -o process\_memory process\_memory.c $(CFLAGS)**

**utils.o : utils.c utils.h**

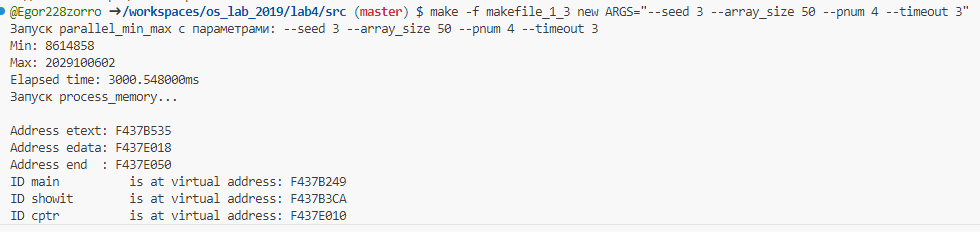
**$(CC) -o utils.o -c utils.c $(CFLAGS)**

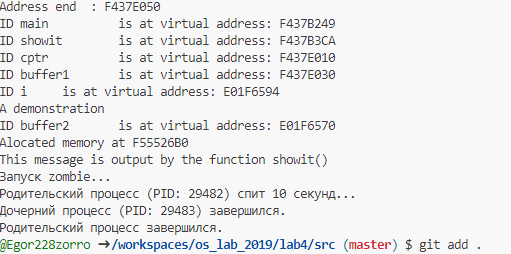
**find\_min\_max.o : find\_min\_max.c utils.h find\_min\_max.h**

**$(CC) -o find\_min\_max.o -c find\_min\_max.c $(CFLAGS)**

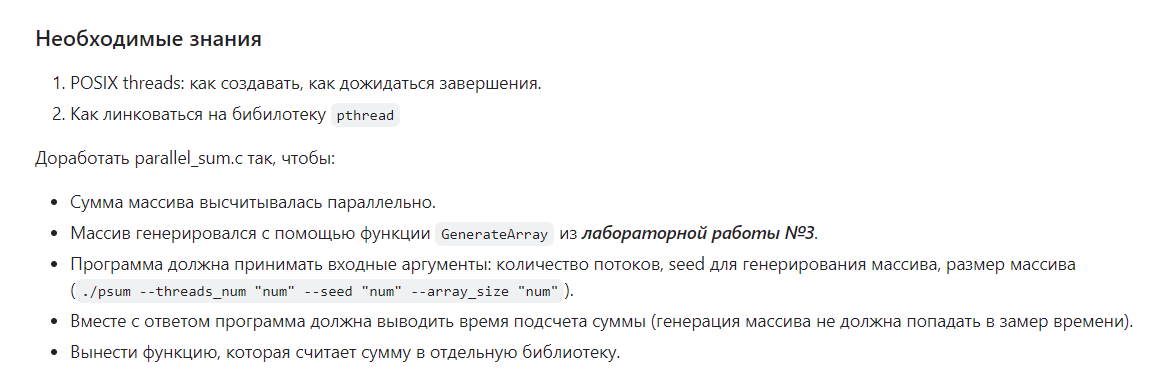
**clean :**

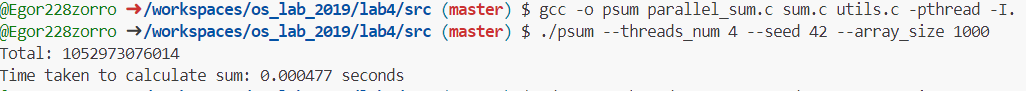
**rm -f utils.o find\_min\_max.o parallel\_min\_max zombie process\_memory**

****

****

**Заданиe 5**

****

****

**#include <stdint.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <stdbool.h>**

**#include "utils.h"**

**#include "sum.h"**

**#include <pthread.h>**

**#include <getopt.h>**

**void \*ThreadSum(void \*args) {**

**struct SumArgs \*sum\_args = (struct SumArgs \*)args;**

**unsigned long long int \*result = malloc(sizeof(unsigned long long int));**

**\*result = Sum(sum\_args);**

**return (void \*)result;**

**}**

**int main(int argc, char \*\*argv) {**

**uint32\_t threads\_num = -1;**

**uint32\_t seed = -1;**

**uint32\_t array\_size = -1;**

**while (true) {**

**int current\_optind = optind ? optind : 1;**

**static struct option options[] = {**

**{"threads\_num", required\_argument, 0, 0},**

**{"seed", required\_argument, 0, 0},**

**{"array\_size", required\_argument, 0, 0},**

**{0, 0, 0, 0}**

**};**

**int option\_index = 0;**

**int c = getopt\_long(argc, argv, "", options, &option\_index);**

**if (c == -1) break;**

**switch (c) {**

**case 0:**

**switch (option\_index) {**

**case 0:**

**threads\_num = atoi(optarg);**

**if (threads\_num <= 0) {**

**printf("threads\_num must be a positive number\n");**

**return 1;**

**}**

**break;**

**case 1:**

**seed = atoi(optarg);**

**if (seed <= 0) {**

**printf("seed must be a positive number\n");**

**return 1;**

**}**

**break;**

**case 2:**

**array\_size = atoi(optarg);**

**if (array\_size <= 0) {**

**printf("array\_size must be a positive number\n");**

**return 1;**

**}**

**break;**

**default:**

**printf("Index %d is out of options\n", option\_index);**

**}**

**break;**

**case '?':**

**break;**

**default:**

**printf("getopt returned character code 0%o?\n", c);**

**}**

**}**

**if (optind < argc) {**

**printf("Has at least one no option argument\n");**

**return 1;**

**}**

**if (seed == -1 || array\_size == -1 || threads\_num == -1) {**

**printf("Usage: %s --seed \"num\" --array\_size \"num\" --pnum \"num\" --timeout \"num\"\n",**

**argv[0]);**

**return 1;**

**}**

**pthread\_t threads[threads\_num];**

**int \*array = malloc(sizeof(int) \* array\_size);**

**GenerateArray(array, array\_size, seed);**

**struct SumArgs args[threads\_num];**

**int chunk\_size = array\_size / threads\_num;**

**clock\_t start\_time = clock();**

**for (uint32\_t i = 0; i < threads\_num; i++) {**

**args[i].array = array;**

**args[i].begin = i \* chunk\_size;**

**args[i].end = (i + 1) \* chunk\_size;**

**if (i == threads\_num - 1) {**

**args[i].end = array\_size;**

**}**

**if (pthread\_create(&threads[i], NULL, ThreadSum, (void \*)&args[i])) {**

**printf("Error: pthread\_create failed!\n");**

**return 1;**

**}**

**}**

**unsigned long long int total\_sum = 0;**

**for (uint32\_t i = 0; i < threads\_num; i++) {**

**unsigned long long int \*sum;**

**pthread\_join(threads[i], (void \*\*)&sum);**

**total\_sum += \*sum;**

**//free(sum);**

**}**

**clock\_t end\_time = clock();**

**double time\_taken = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;**

**free(array);**

**printf("Total: %llu\n", total\_sum);**

**printf("Time taken to calculate sum: %.6f seconds\n", time\_taken);**

**return 0;**

**}**

**POSIX Threads (Pthreads)**

**Pthreads — это библиотека для работы с потоками в POSIX-совместимых системах (Linux, Unix и т.д.). Она предоставляет API для создания, управления потоками, а также синхронизации между ними.**

**1.Как создавать потоки?**

**Для создания потоков используется функция pthread\_create().**

**Прототип pthread\_create:**

**int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start\_routine)(void \*), void \*arg);**

**pthread\_t \*thread: Указатель на идентификатор потока, который будет установлен при его создании.**

**const pthread\_attr\_t \*attr: Атрибуты потока (можно передать NULL для атрибутов по умолчанию).**

**void \*(\*start\_routine)(void \*): Функция, которую будет выполнять поток.**

**void \*arg: Аргумент, передаваемый в функцию потока.**

**2.Как дожидаться завершения потоков?**

**Для ожидания завершения потока используется функция pthread\_join,которая блокирует выполняет,пока поток не завершится.**

**Прототип pthread\_join:**

**int pthread\_join(pthread\_t thread, void \*\*retval);**

**pthread\_t thread: Идентификатор потока, который нужно подождать.**

**void \*\*retval: Указатель для получения значения, возвращенного потоком (или NULL, если значение не нужно).**

**3. Как линковаться с библиотекой pthread**

**При компиляции нужно добавить флаг -pthread**

**gcc program.c –o program -pthread**

**Программа выполняет вычисление суммы элементов массива с использованием потоков POSIX (Pthreads) для параллельной обработки. Рассмотрим, что происходит на каждом этапе, и разберем, как создаются и управляются потоки.**

**Основные этапы программы:**

**1. Обработка аргументов командной строки**

**Программа принимает три аргумента:**

**--threads\_num — количество потоков;**

**--seed — начальное значение для генерации случайных чисел;**

**--array\_size — размер массива.**

**Функция getopt\_long обрабатывает аргументы:**

**Если аргумент некорректен (например, отрицательное значение), выводится ошибка, и программа завершает выполнение.**

**Если все параметры переданы корректно, их значения сохраняются в переменные threads\_num, seed, array\_size.**

**2. Генерация массива**

**После обработки аргументов создается массив целых чисел с использованием выделения памяти:**

**int \*array = malloc(sizeof(int) \* array\_size);**

**Функция GenerateArray заполняет массив случайными числами на основе переданного значения seed.**

**3. Инициализация потоков**

**Для вычисления суммы элементов массива используются потоки.**

**3.1. Создание потоков**

**Потоки создаются с помощью pthread\_create:**

**pthread\_create(&threads[i], NULL, ThreadSum, (void \*)&args[i]);**

**Аргументы:**

**&threads[i]: указатель на идентификатор потока.**

**NULL: атрибуты потока (по умолчанию).**

**ThreadSum: функция, выполняемая в потоке.**

**(void \*)&args[i]: указатель на структуру с параметрами для функции ThreadSum.**

**3.2. Разделение задач между потоками**

**Массив делится на равные части (chunk\_size), которые обрабатывают потоки:**

**args[i].begin = i \* chunk\_size;**

**args[i].end = (i + 1) \* chunk\_size;**

**Последний поток получает оставшуюся часть массива:**

**if (i == threads\_num - 1) {**

**args[i].end = array\_size;**

**}**

**3.3. Функция потока**

**Каждый поток выполняет функцию ThreadSum, которая вычисляет сумму элементов массива в заданном диапазоне:**

**unsigned long long int \*result = malloc(sizeof(unsigned long long int));**

**\*result = Sum(sum\_args);**

**return (void \*)result;**

**Возвращаемое значение: указатель на сумму текущего потока.**

**4. Ожидание завершения потоков**

**Главный поток ждет завершения всех созданных потоков с помощью pthread\_join:**

**pthread\_join(threads[i], (void \*\*)&sum);**

**Аргументы:**

**threads[i]: идентификатор потока, который нужно подождать.**

**(void \*\*)&sum: указатель на переменную, в которую будет записано возвращаемое значение потока.**

**Сумма, вычисленная потоком, добавляется к общей сумме:**

**total\_sum += \*sum;**

**5. Вычисление времени выполнения**

**Используются функции clock для измерения времени выполнения:**

**clock\_t start\_time = clock();**

**...**

**clock\_t end\_time = clock();**

**double time\_taken = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;**

**6. Вывод результата и очистка памяти**

**Выводится общая сумма элементов массива и время выполнения:**

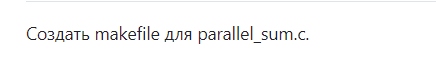
**printf("Total: %llu\n", total\_sum);**

**printf("Time taken to calculate sum: %.6f seconds\n", time\_taken);**

**Освобождается выделенная память:**

**free(array);**

**Задание 6**

****

**CC = gcc**

**CFLAGS = -I.**

**new: parallel\_min\_max zombie process\_memory psum**

**parallel\_min\_max: parallel\_min\_max.c utils.o find\_min\_max.o utils.h find\_min\_max.h**

**$(CC) -o parallel\_min\_max utils.o find\_min\_max.o parallel\_min\_max.c $(CFLAGS)**

**zombie: zombie.c**

**$(CC) -o zombie zombie.c $(CFLAGS)**

**process\_memory: process\_memory.c**

**$(CC) -o process\_memory process\_memory.c $(CFLAGS)**

**psum: parallel\_sum.o sum.o utils.o sum.h utils.h**

**$(CC) -o psum parallel\_sum.o sum.o utils.o -lpthread $(CFLAGS)**

**utils.o: utils.c utils.h**

**$(CC) -c utils.c $(CFLAGS)**

**find\_min\_max.o: find\_min\_max.c find\_min\_max.h utils.h**

**$(CC) -c find\_min\_max.c $(CFLAGS)**

**sum.o: sum.c sum.h**

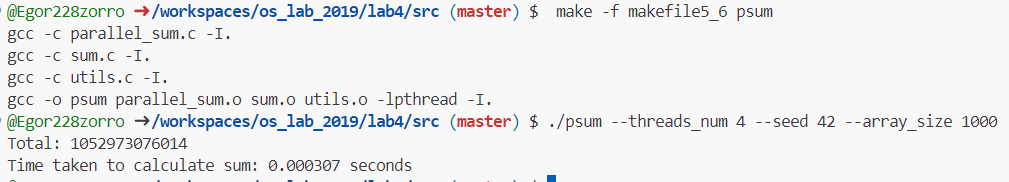
**$(CC) -c sum.c $(CFLAGS)**

**parallel\_sum.o: parallel\_sum.c sum.h utils.h**

**$(CC) -c parallel\_sum.c $(CFLAGS)**

**clean:**

**rm -f utils.o find\_min\_max.o parallel\_min\_max zombie process\_memory parallel\_sum.o sum.o psum**

****

**Ссылка на githab:** **https://github.com/Egor228zorro/os\_lab\_2019.git**