Лабораторная работа №4

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.

```
@Egor228zorro ➜/workspaces/os_lab_2019/lab4/src (master) $ make
gcc -c utils.c -I.
gcc -c find_min_max.c -I.
gcc -c parallel_min_max.c -I.
gcc -o parallel_min_max utils.o find_min_max.o parallel_min_max.o -I.
@Egor228zorro ➜/workspaces/os_lab_2019/lab4/src (master) $ ./parallel_min_max --seed 50 --array_size 50 --pnum 4 --timeout 5
Timeout reached. Killing child processes.
Min: 127574857
Max: 1571951721
Elapsed time: 5000.563000ms
```

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.

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.

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.

```
@Egor228zorro ➜/workspaces/os_lab_2019/lab4/src (master) $ cat >zombie.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    pid_t pid;

    // Создание дочернего процесса
    if ((pid = fork()) == 0) {
        // Дочерний процесс
        printf("Дочерний процесс (PID: %d) завершился.\n", getpid());
```

```
@Egor228zorro ➜/workspaces/os_lab_2019/lab4/src (master) $
@Egor228zorro ➜/workspaces/os_lab_2019/lab4/src (master) $ gcc -o zombie zombie.c
@Egor228zorro ➜/workspaces/os_lab_2019/lab4/src (master) $ chmod +x zombie
@Egor228zorro ➜/workspaces/os_lab_2019/lab4/src (master) $ ./zombie
Родительский процесс (PID: 7958) спит 10 секунд...
Дочерний процесс (PID: 7959) завершился.
Родительский процесс завершился.
```

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().

Задание 3

Необходимые знания

1. Работа виртуальной памяти.

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

_

```
@Egor228zorro ➜/workspaces/os_lab_2019/lab4/src (master) $  ./process_memory

Address etext: 2E7CA535
Address edata: 2E7CD018
Address end  : 2E7CD050
ID main          is at virtual address: 2E7CA249
ID showit        is at virtual address: 2E7CA3CA
ID cptr          is at virtual address: 2E7CD010
ID buffer1       is at virtual address: 2E7CD030
ID i     is at virtual address: 7E55EC24
A demonstration
ID buffer2       is at virtual address: 7E55EC00
Alocated memory at 2FC826B0
This message is output by the function showit()
```

B 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.

Задание 4

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

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
```

```
@Egor228zorro →/workspaces/os_lab_2019/lab4/src (master) $ make -f makefile_1_3 new ARGS="--seed 3 --array_size 50 --pnum 4 --timeout 3"
Запуск parallel_min_max с параметрами: --seed 3 --array_size 50 --pnum 4 --timeout 3
Min: 8614858
Max: 2029100602
Elapsed time: 3000.548000ms
Запуск process_memory...

Address etext: F437B535
Address edata: F437E018
Address end  : F437E050
ID main         is at virtual address: F437B249
ID showit       is at virtual address: F437B3CA
ID cptr         is at virtual address: F437E010
```

```
Address end  : F437E050
ID main          is at virtual address: F437B249
ID showit        is at virtual address: F437B3CA
ID cptr          is at virtual address: F437E010
ID buffer1       is at virtual address: F437E030
ID i     is at virtual address: E01F6594
A demonstration
ID buffer2       is at virtual address: E01F6570
Alocated memory at F55526B0
This message is output by the function showit()
Запуск zombie...
Родительский процесс (PID: 29482) спит 10 секунд...
Дочерний процесс (PID: 29483) завершился.
Родительский процесс завершился.
@Egor228zorro ➜/workspaces/os_lab_2019/lab4/src (master) $ git add .
```

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.

```
@Egor228zorro →/workspaces/os_lab_2019/lab4/src (master) $ gcc -o psum parallel_sum.c sum.c utils.c -pthread -I.
@Egor228zorro →/workspaces/os_lab_2019/lab4/src (master) $ ./psum --threads_num 4 --seed 42 --array_size 1000
Total: 1052973076014
Time taken to calculate sum: 0.000477 seconds
```

```
#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

Создать makefile для parallel_sum.c.

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

```
@Egor228zorro ➜/workspaces/os_lab_2019/lab4/src (master) $  make -f makefile5_6 psum
gcc -c parallel_sum.c -I.
gcc -c sum.c -I.
gcc -c utils.c -I.
gcc -o psum parallel_sum.o sum.o utils.o -lpthread -I.
@Egor228zorro ➜/workspaces/os_lab_2019/lab4/src (master) $ ./psum --threads_num 4 --seed 42 --array_size 1000
Total: 1052973076014
Time taken to calculate sum: 0.000307 seconds
```

Ссылка на github: https://github.com/Egor228zorro/os_lab_2019.git