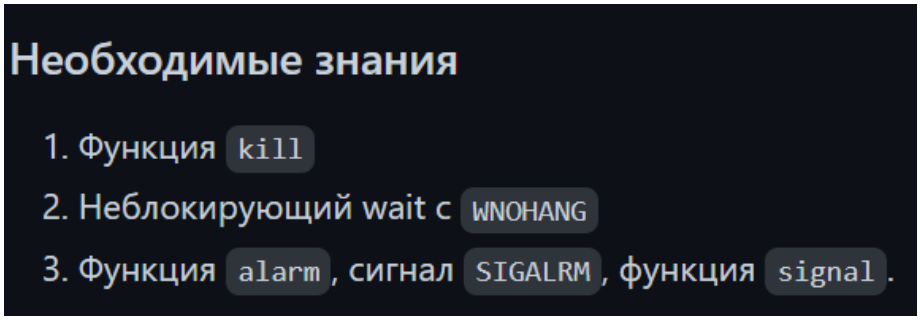
Лабораторная работа № 4.

Немецков Михаил, ПМ-31

Задание 1.



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) Команда **kill** — это встроенная команда, которая используется для ручного завершения процессов. Команда *kill* отправляет сигнал процессу, который завершает процесс. Если пользователь не указывает какой-либо сигнал, который должен быть отправлен вместе с командой kill, то отправляется сигнал *TERM по* умолчанию, который завершает процесс.

2) В **(wait)** ожидании () системный вызов приостанавливает выполнение вызывающего процесса до момента, пока не прекратится процесс одного из своих детей. WNOHANG -немедленно вернуться, если ни один ребенок не вышел.

3) **alarm** () (это функция для подачи сигнала) организует доставку сигнала SIGALRM вызывающему процессу в считанные секунды. Если секунды равны нулю, любой ожидающий сигнал тревоги отменяется. В любом случае любой ранее установленный сигнал тревоги () отменяется.

Функция **signal** (для обработки сигналов). Сигнал () задает расположение сигнатуры сигнала к обработчику, который является либо SIG_IGN , SIG_DFL , либо адресом программиста-определенная функция («обработчик сигнала»). Если сигнал сигнум доставляется в процессе, то происходит следующее:

* Если расположение установлено на SIG_IGN , то сигнал игнорируется.

* Если для диспозиции установлено значение SIG_DFL, то действие по умолчанию.

* Если диспозиция установлена на функцию, то сначала либо disposition сбрасывается в SIG_DFL или сигнал блокируется, а затем вызывается обработчик с аргументом signum. Если вызов обработчика привел к тому, что сигнал заблокирован, то сигнал разблокируется при возврате к обработчику.

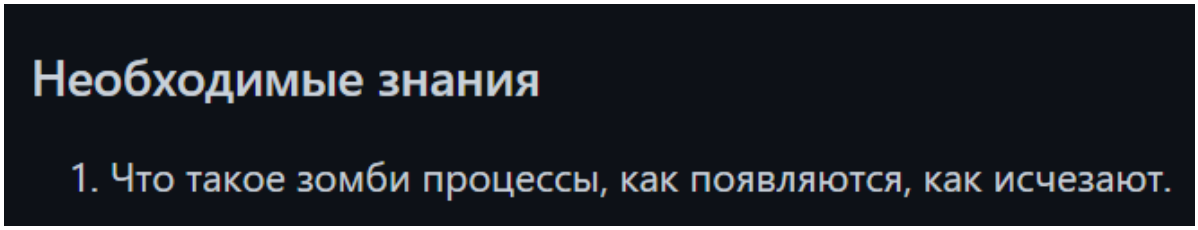
Дополнить программу parallel_min_max.c

(см. Гитхаб)

~/lab4/src$./parallel_min_max.exe —seed 100 —array_size 1000000000 —pnum 2 —timeout 1
Timeout is over.
Process 9340 was slain :(
Process 9341 was slain :(

(мы задали 2 процесса, родитель не дождался времени заданного и убил два процесса дочерних)

Задание 2.



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?

Процесс-зомби— дочерний процесс в Unix-системе, завершивший своё выполнение, но ещё присутствующий в списке процессов операционной системы, чтобы дать родительскому процессу считать код завершения.

Как появляются зомби процессы?

Процесс при завершении освобождает все свои ресурсы (за исключением **PID** — идентификатора процесса) и становится «зомби» — пустой записью в таблице процессов, хранящей код завершения для родительского процесса.

Система уведомляет родительский процесс о завершении дочернего с помощью сигнала **SIGCHLD**. Предполагается, что после получения **SIGCHLD** он считает код возврата с помощью системного вызова **wait()**, после чего запись зомби будет удалена из списка процессов. Если родительский процесс игнорирует **SIGCHLD** (а он игнорируется по умолчанию), то зомби остаются до его завершения.

Как исчезают зомби процессы?

Убить зомби процесс на прямую командой kill нельзя. Вариант полностью убрать зомби процесс — убить или перезапустить его родительский процесс.

Что бы убить зомби процесс, для начала нужно получить его PID. Потом получить PID родительского процесса. Когда мы знаем PID, правильнее всего определить, что это за процесс командой top, а затем перезапустить его. В крайнем случае можно убить родительский процесс командой kill.

Чем опасны зомби процессы?

Однако каждый зомби-процесс сохраняет свой идентификатор процесса (PID). Системы Linux имеют ограниченное количество идентификаторов процессов - 32767 по умолчанию в 32-битных системах. Если зомби накапливаются очень быстро - например, если неправильно запрограммированное серверное программное обеспечение создает зомби-процессы под нагрузкой - весь пул доступных PID в конечном итоге будет назначен процессам-зомби, предотвращая запуск других процессов.

Программа (см. Гитхаб)

```
#include
<stdio.h>
            #include <unistd.h>
            int main(int argc, char* argv[])
            {
                pid_t pid;
                pid = fork();
                if(pid > 0)
                {
                    printf("Hello! I am main process. I have 1 chiled with pid %d\n",pid);
                    sleep(100);
                }
                else
                    return 1;
                return 0;
            }
```

(pid_t –примитивный тип данных, который определяет идентификатор процесса. При вызове fork() порождается новый процесс (процесс-потомок), который почти идентичен порождающему процессу-родителю)

~/lab4/src$ ./Shawn.exe

Hello! I am main process. I have 1 chiled with pid 1686

Программа заснула

Задание 3.

Необходимые знания

1. Работа виртуальной памяти.

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

1)***etext-*** Это первый адрес после конца текстового сегмента (программного кода).

edata- Это первый адрес после конца инициализированного сегмента данных.

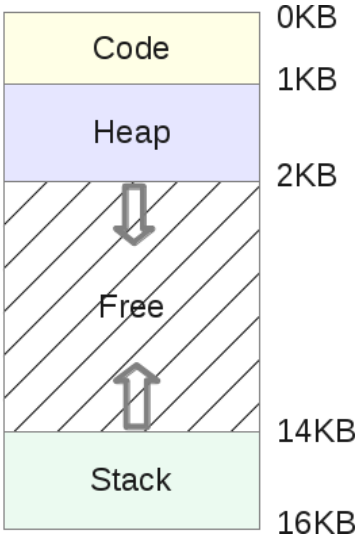
end*(конец)*- Это первый адрес после конца неинициализированного сегмента данных (также известного как сегмент BSS).

2) **работа с виртуальной памятью**

Адресное пространство

Это некая абстракция памяти, выделяемая ОС для какого-то процесса. Адресное пространство содержит всё, что нужно для выполнения процесса:

- Машинные инструкции, которые должен выполнить ЦПУ.
- Данные, с которыми будут работать эти машинные инструкции.



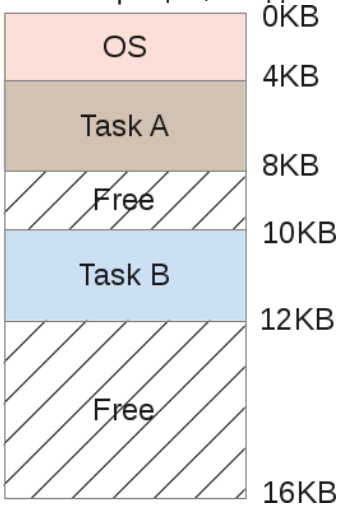
- Стек (stack) — это область памяти, в которой программа хранит информацию о вызываемых функциях, их аргументах и каждой локальной переменной в

функциях. Размер области может меняться по мере работы программы. При вызове функций стек увеличивается, а при завершении — уменьшается.
- Куча (heap) — это область памяти, в которой программа может делать всё, что заблагорассудится. Размер области может меняться. Программист имеет возможность воспользоваться частью памяти кучи с помощью функции `malloc()`, и тогда эта область памяти увеличивается. Возврат ресурсов осуществляется с помощью `free()`, после чего куча уменьшается.
- Кодовый сегмент (code) — это область памяти, в которой хранятся машинные инструкции скомпилированной программы. Они генерируются компилятором, но могут быть написаны и вручную. Обратите внимание, что эта область памяти также может быть разделена на три части (текст, данные и BSS). Эта область памяти имеет фиксированный размер, определяемый компилятором. В нашем примере пусть это будет 1 Кб.

Виртуализация памяти

Допустим, задача A получила в своё распоряжение всю доступную пользовательскую память. И тут возникает задача B. Как быть? Решение было найдено в виртуализации.

Иллюстрация, когда в памяти одновременно находятся A и B:



Допустим, A пытается получить доступ к памяти в собственном адресном пространстве, например по индексу 11 Кб. Возможно даже, что это будет её собственный стек. В этом случае ОС нужно придумать, как не подгружать индекс 1500, поскольку по факту он может указывать на область задачи B.
На самом деле, адресное пространство, которое каждая программа считает своей памятью, является **памятью виртуальной**. *Фальшивкой*. И в области памяти задачи A индекс 11 Кб будет фальшивым адресом. То есть — адресом виртуальной памяти.
Каждая программа, выполняющаяся на компьютере, работает с фальшивой (виртуальной) памятью. С помощью некоторых чипов ОС обманывает процесс, когда он обращается к какой-либо области памяти. Благодаря виртуализации ни один процесс не может получить доступ к памяти, которая ему не принадлежит: задача A не влезет в память задачи B или самой ОС. При этом на пользовательском уровне всё абсолютно прозрачно,

благодаря обширному и сложному коду ядра ОС.
Таким образом, каждое обращение к памяти регулируется операционной системой.

Переадресация (транслирование, перевод, преобразование адресов) — это термин, обозначающий процесс сопоставления виртуального адреса физическому.

Система виртуальной памяти в Linux поддерживает адресное пространство, видимое каждому процессу: она создает страницы виртуальной памяти по требованию и управляет загрузкой этих страниц с диска или откачкой их обратно на диск, если требуется. Менеджер виртуальной памяти поддерживает две точки зрения на адресное пространство каждого процесса:

· **Логическую**– поддержка команд управления адресным пространством. Адресное пространство рассматривается как совокупность непересекающихся смежных областей.

· **Физическую**– с помощью таблицы страниц для каждого процесса.

Ядро создает новое виртуальное адресное пространство:

· Когда процесс запускает новую программу системным вызовом exec;

· При создании нового процесса системным вызовом fork.

При исполнении новой программы процессу предоставляется новое, пустое адресное пространство; процедуры загрузки программ наполняют это адресное пространство регионами виртуальной памяти. Создание нового процесса с помощью fork включает создание полной копии адресного пространства существующего процесса. Ядро копирует дескрипторы доступа к виртуальной памяти родительского процесса, затем создает новый набор таблиц страниц для дочернего процесса. Таблицы страниц процесса-родителя копируются непосредственно в таблицы страниц дочернего, причем счетчик ссылок на каждую страницу увеличивается. После исполнения fork родительский и дочерний процесс используют одни и те же физические страницы в своих виртуальных адресных пространствах. Система управления страницами откачивает страницы физической памяти на диск, если они требуются для какой-либо другой цели. Система управления страницами делится на две части:

· Алгоритм откачки, который определяет, какие страницы и когда откачать на диск;

· Механизм подкачки фактически выполняет передачу и подкачивает данные обратно в физическую память, если требуется.

Компиляция программы

~/lab4/src$ ./process_memory.exe

Address etext: f9d81535

Address edata: f9d84018

Address end : f9d84050

ID main   is at virtual address: f9d81249

ID showit   is at virtual address: f9d813c5

ID cptr   is at virtual address: f9d84010

ID buffer1   is at virtual address: f9d84030

ID i   is at virtual address: be2a2574

A demonstration

ID buffer2   is at virtual address: be2a2550

Alocated memory at faf996b0

This message is output by the function showit()

(переменные cptr и buffer1 лежат рядом в памяти, т.к так по коду, и у них вирт.адреса отличаются на 00000020, что соответствует 4 байтам в 16-ричной системе счисления, а это есть 32 бита. Все переменные объявлены глобально, а переменная i объявлена локально, потому ее адрес сильно отличается от адресов других переменных)

Задание 4.

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

(см.Гитхаб)

Задание 5.

Необходимые знания

1. POSIX threads: как создавать, как дожидаться завершения.

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.

Библиотеки POSIX threads

Позволяют запускать новый параллельный поток процессов. Цель использования - ускорить выполнение программы.

Пример создания потока (плюс сразу его исполняем)

```
status = pthread_create(&thread, NULL, helloWorld, NULL);
if (status != 0) {
    printf("main error: can't create thread, status = %d\n", status);
    exit(ERROR_CREATE_THREAD);
}
```

```
// Это функция, которая будет работать в отдельном потоке. Она не будет получать никаких аргументов

void* helloWorld(void *args) {
    printf("Hello from thread!\n");
    return SUCCESS;
}
```

Вызов потока

```
status = pthread_join(thread, (void**)&status_addr);
if (status != SUCCESS) {
    printf("main error: can't join thread, status = %d\n", status);
    exit(ERROR_JOIN_THREAD);
}
```

Приводит к тому, что основной поток будет ждать завершения порождённого.

Функция
`int pthread_join(pthread_t thread, void **value_ptr);`
Откладывает выполнение вызывающего (эту функцию) потока, до тех пор, пока не будет выполнен поток thread. Когда pthread_join выполнилась успешно, то она возвращает 0. Если поток явно вернул значение (это то самое значение SUCCESS, из нашей функции), то оно будет помещено в переменную value_ptr. Возможные ошибки, которые возвращает pthread_join

См.Гитхаб

(здесь только мои дополнения)

```
pthread_t
threads[threads_num];
```

```
                        int *array = (int*)calloc(array_size,sizeof(int));
                        GenerateArray(array, array_size, seed);

                        struct timeval start_time;
                        gettimeofday(&start_time, NULL);


                        struct SumArgs args[threads_num];
                        args[0].begin = 0;


                        for (uint32_t i = 0; i < threads_num; i++)
                        {
                            args[i].array = array;
                              if (i != 0)
                                  args[i].begin = args[i - 1].end;
                              if (args[i].begin == array_size)
                                  break;
                              if (i != threads_num - 1) {
                                  args[i].end = (i + 1) * array_size / threads_num;
                              } else {
                                  args[i].end = array_size;
                              }

                            if (pthread_create(&threads[i], NULL, ThreadSum, (void *)(args
                    + i)) != 0) {
                              printf("Error: pthread_create failed!\n");
                              return 1;
                            }
                          }

                          intmax_t total_sum = 0;
                          for (uint32_t i = 0; i < threads_num; i++) {
                            intmax_t sum = 0;
                            pthread_join(threads[i], (void **)&sum);
                            total_sum += sum;
                          }

                          free(array);

                          struct timeval finish_time;
                          gettimeofday(&finish_time, NULL);
```

```
  double elapsed_time = (finish_time.tv_sec - start_time.tv_sec) *
1000.0;
  elapsed_time += (finish_time.tv_usec - start_time.tv_usec) /
1000.0;

  printf("Total: %d\n", total_sum);
  printf("Elapsed time: %fms\n", elapsed_time);

  return 0;
}
```

~/lab4/src$ ./parallel_sum --threads_num 4 --seed 1 --array_size 100

Total: 884

Elapsed time: 1.052000ms

Задание 6.

Создать makefile для parallel_sum.c.

(См.Гитхаб)