OS2 Lab9

checkbox	☑
due date	@October 1, 2021
class	OS
Status	approved

Task

Модифицируйте программу упражнения 8 так, чтобы сама по себе она не завершалась. Вместо этого, после нажания ^-C (после получения сигнала SIGINT) программа должна как можно скорее завершаться, собирать частичные суммы ряда и выводить полученное приближение числа.

Рекомендации: ожидаемое решение состоит в том, что вы установите обработчик SIGINT. Этот обработчик должен устанавливать глобальную флаговую переменную. Вычислительные нити должны просматривать значение флага через некоторое количество итераций, например через 1000000, и выходить при помощи pthread_exit, если флаг установлен. Подумайте, как минимизировать ошибку, обусловленную тем, что разные потоки к моменту завершения успели пройти разное количество итераций. Скорее всего, такая минимизация должна обеспечиваться за счет увеличения времени между получением сигнала и выходом.

Notes

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <signal.h>
#include <stdbool.h>

#define ITERATION_STEP 2000

bool exitFlag = false;
pthread_mutex_t mutex;
int threadNum = 1;
pthread_barrier_t barrier;

int maxIterationReached;

// thread's arguments

typedef struct ThreadWork{
    int initialIndex; // number of iteration to start with
    double partialSum;
} ThreadWork;

void* countPartialSum(void* args){
    ThreadWork* work = (ThreadWork *)args;
    double partialSum = 0;
    int stepNumber = work->initialIndex;
    long int iterationCount = 0;

    for (int i = stepNumber; ; i += threadNum) {
        partialSum += 1.0 / (i * 4.0 + 1.0);
        partialSum -= 1.0 / (i * 4.0 + 3.0);
        iterationCount++;

        if (iterationCount % ITERATION_STEP == 0){
            pthread_barrier_wait(&barrier);
            if (exitFlag || iterationCount < maxIterationReached){
                printf("on thread %lu exit flag is true. Unlocking mutex and terminating. ITER COU
NT = %d\n", pthread_self(), iterationCount);
                work->partialSum = partialSum * 4;
                pthread_exit(NULL);
            } else {
                stepNumber += threadNum * ITERATION_STEP;
                pthread_mutex_lock(&mutex);
                if (iterationCount > maxIterationReached){
                    maxIterationReached = iterationCount;
                }
```

Мутекс является одним из таких примитивов. Слово мутекс (mutex) происходит от сокращения словосочетания mutual exclusion - взаимное исключение. В некоторых русскоязычных публикациях эти объекты также называют мьютексами. Такая транскрипция ближе к правильному английскому произношению этого слова.

Мутекс может находиться в двух состояниях - свободном и захваченном. Над мутексом определены две основные операции - блокировка (lock, захват, asquire) и снятие (unlock, освобождение, release). Блокировка свободного мутекса приводит к его переводу в захваченное состояние. Попытка блокировки захваченного мутекса приводит к засыпанию (блокировке) нити, которая пыталась выполнить эту операцию. Освобождение свободного мутекса - недопустимая операция; в зависимости от особенностей реализации эта операция может приводить к непредсказуемым последствиям или к ошибке или просто игнорироваться. Освобождение занятого мутекса приводит к переводу мутекса в свободное состояние; если в этот момент на мутексе были заблокированы одна или несколько нитей, одна из этих нитей пробуждается и захватывает мутекс.

Применение мутексов для решения задачи взаимоисключения очевидно. Мы должны связать с каждым разделяемым ресурсом мутекс. Когда нить входит в критическую секцию, связанную с ресурсом, она должна захватить мутекс, а когда выходит из нее - освободить.

Мутекс — примитив синхронизации.

Примитив — это переменная непрозрачного типа, над которой определен некоторый набор операций. Вы можете использовать эти операции, но вам не следует работать с объектом мимо этих операций. (например, pthread — примитив)

Перед освобождением памяти из-под мутекса мутекс необходимо уничтожить. Это делается функцией `pthread_mutex_destroy(3C)`. Операции над мутексом могут приводить к размещению дополнительной памяти или объектов ядра ОС, поэтому уничтожение мутекса без выполнения `pthread_mutex_destroy` может приводить к утечке памяти или исчерпанию системных ресурсов. Выполнение операции `pthread_mutex_destroy` над мутексом, на котором заблокирована одна или несколько нитей, приводит к неопределенным последствиям. Дальнейшие операции над этим мутексом также приводят к неопределенным последствиям.

```
                pthread_mutex_unlock(&mutex);
            }
        }
    }

}

void signalHandler(int signal){
    exitFlag = true;
}

int main(int argc, char* argv[]){
    if (argc != 2){
        printf("This program calculates Pi using Leibniz series.\n"
               "Please pass it thread number to start calculation.\n");
        exit(-1);
    }
    sigset_t maskWithBlockedSIGINT;
    sigset_t defaultMask;

    sigaddset(&maskWithBlockedSIGINT, SIGINT);

    signal(SIGINT, signalHandler);
    pthread_sigmask(SIG_SETMASK, &maskWithBlockedSIGINT, &defaultMask);
    pthread_mutex_init(&mutex, NULL);


    threadNum = atoi(argv[1]);

    if (threadNum < 1){
        printf("This program calculates Pi using Leibniz series.\n"
               "Please pass it thread number to start calculation.\n");
        exit(-1);
    }

    if (pthread_barrier_init(&barrier, NULL, threadNum) != 0) {
        perror("failed to init barrier");
        exit(-1);
    }

    pthread_t* threads = calloc(threadNum, sizeof(pthread_t *));
    ThreadWork* work = calloc(threadNum, sizeof(ThreadWork));
    time_t start,end;
    time (&start);

    for (int i = 0; i < threadNum; i++){

        work[i].initialIndex = i;

        if (pthread_create(&threads[i], NULL, countPartialSum, &work[i]) != 0 ){
            perror("failed to create new thread");
            free(threads);
            free(work);
            exit(-1);
        }
    }

    pthread_sigmask(SIG_SETMASK, &defaultMask, &maskWithBlockedSIGINT);

    double pi = 0;

    for (int i = 0; i < threadNum; i++){
        if (pthread_join(threads[i], NULL) != 0 ){
            perror("failed to join thread");
            free(threads);
            free(work);
            exit(-1);
        }

        pi += work[i].partialSum;
    }
    time (&end);
    double dif = difftime (end,start);
    printf("pi = %.14f\n", pi);
    printf ("Your calculations took %.10lf seconds to run.\n", dif );

    pthread_mutex_destroy(&mutex);
    pthread_barrier_destroy(&barrier);
    free(threads);
    free(work);
    return 0;
}
```

Тип `ERRORCHECK` требует, чтобы все операции над мутексами проверяли состояние мутекса и возвращали ошибки при недопустимых последовательностях операций над мутексом. Из описания `pthread_attr_settype(3C)` можно сделать вывод, что такие мутексы делают проверку на мертвую блокировку с участием нескольких нитей, но это не так. В соответствии с требованиями стандарта, код ошибки `EDEADLK` возвращается только при попыткезахвата мутекса, уже занятого текущейнитью. Мутексытипа `ERRORCHECK` удобныдля отладки приложений, но требуют гораздо большего объема вычислений, чем мутексы типа `NORMAL`.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <signal.h>
#include <stdbool.h>

#define ITERATION_STEP 2000

bool exitFlag = false;
pthread_mutex_t mutex;
int threadNum = 1;
pthread_barrier_t barrier;

// thread's arguments

typedef struct ThreadWork{
    int initialIndex; // number of iteration to start with
    double partialSum;
} ThreadWork;

void* countPartialSum(void* args){
    ThreadWork* work = (ThreadWork *)args;
    double partialSum = 0;
    int stepNumber = work->initialIndex;
    long int iterationCount = 0;

    for (int i = stepNumber; ; i += threadNum) {
        partialSum += 1.0 / (i * 4.0 + 1.0);
        partialSum -= 1.0 / (i * 4.0 + 3.0);
        iterationCount++;

        if (iterationCount % ITERATION_STEP == 0){
            pthread_mutex_lock(&mutex);
            if (exitFlag){
                printf("on thread %lu exit flag is true. Unlocking mutex and terminating", pthread
_self());
                work->partialSum = partialSum * 4;
                pthread_mutex_unlock(&mutex);
                pthread_barrier_wait(&barrier);
                pthread_exit(NULL);
            } else {
                stepNumber += threadNum * ITERATION_STEP;
                pthread_mutex_unlock(&mutex);
            }
        }
    }

}

void signalHandler(int signal){
    pthread_mutex_lock(&mutex);
    exitFlag = true;
    pthread_mutex_unlock(&mutex);
}

int main(int argc, char* argv[]){
    if (argc != 2){
        printf("This program calculates Pi using Leibniz series.\n"
               "Please pass it thread number to start calculation.\n");
        exit(-1);
    }
    sigset_t maskWithBlockedSIGINT;
    sigset_t defaultMask;

    sigaddset(&maskWithBlockedSIGINT, SIGINT);

    signal(SIGINT, signalHandler);
    pthread_sigmask(SIG_SETMASK, &maskWithBlockedSIGINT, &defaultMask);
    pthread_mutex_init(&mutex, NULL);


    threadNum = atoi(argv[1]);

    if (threadNum < 1){
        printf("This program calculates Pi using Leibniz series.\n"
               "Please pass it thread number to start calculation.\n");
        exit(-1);
    }

    if (pthread_barrier_init(&barrier, NULL, threadNum) != 0) {
        perror("failed to init barrier");
        exit(-1);
    }
```

```
    pthread_t* threads = calloc(threadNum, sizeof(pthread_t *));
    ThreadWork* work = calloc(threadNum, sizeof(ThreadWork));
    time_t start,end;
    time (&start);

    for (int i = 0; i < threadNum; i++){

        work[i].initialIndex = i;

        if (pthread_create(&threads[i], NULL, countPartialSum, &work[i]) != 0 ){
            perror("failed to create new thread");
            free(threads);
            free(work);
            exit(-1);
        }
    }

    pthread_sigmask(SIG_SETMASK, &defaultMask, &maskWithBlockedSIGINT);

    double pi = 0;

    for (int i = 0; i < threadNum; i++){
        if (pthread_join(threads[i], NULL) != 0 ){
            perror("failed to join thread");
            free(threads);
            free(work);
            exit(-1);
        }

        pi += work[i].partialSum;
    }
    time (&end);
    double dif = difftime (end,start);
    printf("pi = %.14f\t", pi);
    printf ("Your calculations took %.10lf seconds to run.\n", dif );

    pthread_mutex_destroy(&mutex);
    pthread_barrier_destroy(&barrier);
    free(threads);
    free(work);
    return 0;
}
```

Reading list

s3.us-west-2.amazonaws.com

Поток может приостановиться в ожидании доставки сигнала, вызвав функцию `sigwait`.


```
#include <signal.h>

int sigwait(const sigset_t *restrict set, int *restrict signop);

                    Возвращает 0 в случае успеха, код ошибки — в случае неудачи
```

Аргумент *set* определяет набор сигналов, доставка которых ожидается. По возвращении из функции по адресу *signop* будет записан номер доставленного сигнала.

Если какой-либо сигнал, входящий в набор *set*, ожидал обработки к моменту вызова `sigwait`, функция вернет управление немедленно. Перед возвратом управления `sigwait` удалит этот сигнал из набора сигналов, ожидающих обработки. Если реализация поддерживает очереди сигналов и в очереди находится несколько экземпляров одного и того же сигнала, `sigwait` удалит из очереди только один экземпляр, остальные останутся в очереди, ожидая обработки.

Во избежание ошибочной реакции на сигнал поток должен заблокировать ожидаемые сигналы перед вызовом `sigwait`. Функция `sigwait` атомарно разблокирует сигналы и перейдет в режим ожидания, пока один из сигналов не будет доставлен. Перед возвратом управления `sigwait` восстановит маску сигналов потока. Если сигнал не будет заблокирован к моменту вызова функции, возникнет промежуток времени, когда сигнал может быть доставлен потоку еще до того, как он выполнит вызов `sigwait`.

Преимущество использования функции `sigwait` заключается в том, что она позволяет упростить обработку сигналов и обрабатывать асинхронные сигналы на синхронный манер. Чтобы воспрепятствовать прерыванию выполнения потока по сигналу, можно добавить требуемые сигналы к маске сигналов каждого потока. Благодаря этому можно выделить отдельные потоки, которые будут заниматься только обработкой сигналов. Эти специально выделенные потоки могут обращаться к любым функциям, которые нельзя использовать в обработчиках сигналов, потому что в этой ситуации функции будут вызываться в контексте обычного потока, а не из традиционного обработчика сигнала, прерывающего работу потока.

Если сразу несколько потоков окажутся заблокированными в функции `sigwait` в ожидании одного и того же сигнала, только в одном из них функция `sigwait` вернет управление, когда сигнал будет доставлен процессу. Если сигнал перехватывается процессом (например, когда процесс установил обработчик сигнала с помощью функции `sigaction`) и при этом поток, обратившийся к функции `sigwait`, ожидает доставки того же сигнала, принятие решения о способе доставки сигнала оставляется на усмотрение реализации.

Операционная система в этом случае может либо вызвать установленный обработчик сигнала, либо позволить функции `sigwait` вернуть управление в поток, но не то и другое вместе. Для посылки сигнала процессу используется функция `kill` (раздел 10.9). Для посылки сигнала потоку используется функция `pthread_kill`.

```
#include <signal.h>

int pthread_kill(pthread_t thread, int signo);

                    Возвращает 0 в случае успеха, код ошибки — в случае неудачи
```

s3.us-west-2.amazonaws.com

Операционная система в этом случае может либо вызвать установленный обработчик сигнала, либо позволить функции `sigwait` вернуть управление в поток, но не то и другое вместе. Для посылки сигнала процессу используется функция `kill` (раздел 10.9). Для посылки сигнала потоку используется функция `pthread_kill`.

```
#include <signal.h>

int pthread_kill(pthread_t thread, int signo);
```

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

Передав в аргументе `signo` значение 0, можно проверить существование потока. Если действие по умолчанию для сигнала заключается в завершении процесса, передача такого сигнала потоку приведет к завершению всего процесса.

Обратите внимание, что таймеры являются ресурсами процесса, и все потоки совместно используют один и тот же набор таймеров. Следовательно, в многопоточном приложении невозможно использовать таймеры в одном потоке, не оказывая влияния на другие (это тема упражнения 12.6).

Пример

В программе в листинге 10.16 мы приостанавливали работу процесса, пока обработчик сигнала не установит флаг, который указывает, что процесс должен завершить работу. Единственными потоками управления в этой программе были главный поток программы и обработчик сигнала, поэтому блокировка сигнала служила надежным средством, не позволяющим пропустить получение сигнала и изменение флага. В многопоточном приложении мы вынуждены защищать доступ к флагу с помощью мьютекса, как показано в листинге 12.6.

Листинг 12.6. Синхронная обработка сигнала

```
#include "apue.h"
#include <pthread.h>

int        quitflag; /* поток записывает сюда ненулевое значение */
sigset_t   mask;

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t waitloc = PTHREAD_COND_INITIALIZER;

void *
thr_fn(void *arg)
{
    int err, signo;

    for (;;) {
        err = sigwait(&mask, &signo);
        if (err != 0)
            err_exit(err, "ошибка вызова функции sigwait");
        switch (signo) {
        case SIGINT:
            printf("\nпрерывание\n");
            break;

        case SIGQUIT:
            pthread_mutex_lock(&lock);
            quitflag = 1;
            pthread_mutex_unlock(&lock);
            pthread_cond_signal(&waitloc);
            return(0);

        default:
            printf("получен непредвиденный сигнал %d\n", signo);
            exit(1);
        }
    }
```

s3.us-west-2.amazonaws.com


```
}

int
main(void)
{
    int        err;
    sigset_t   oldmask;
    pthread_t  tid;

    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigaddset(&mask, SIGQUIT);
    if ((err = pthread_sigmask(SIG_BLOCK, &mask, &oldmask)) != 0)
        err_exit(err, "ошибка выполнения операции SIG_BLOCK");

    err = pthread_create(&tid, NULL, thr_fn, 0);
    if (err != 0)
        err_exit(err, "невозможно создать поток");

    pthread_mutex_lock(&lock);
    while (quitflag == 0)
        pthread_cond_wait(&waitloc, &lock);
    pthread_mutex_unlock(&lock);

    /*
     * Сигнал SIGQUIT был перехвачен и к настоящему моменту
     * опять заблокирован.
     */
    quitflag = 0;

    /* Восстановить маску сигналов, в которой SIGQUIT разблокирован. */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("ошибка выполнения операции SIG_SETMASK");
    exit(0);
}
```

Вместо того чтобы обрабатывать сигнал в функции-обработчике, прерывающей выполнение главного потока, мы создали для этого отдельный поток. Изменение флага `quitflag` производится под защитой мьютекса, чтобы главный поток не смог пропустить изменение значения флага, когда поток-обработчик вызывает функцию `pthread_cond_signal`. Тот же самый мьютекс используется в главном потоке для контроля состояния флага, мы атомарно освобождаем его и ожидаем наступления события.

Обратите внимание, что в самом начале главный поток программы блокирует сигналы `SIGINT` и `SIGQUIT`. Когда создается поток, который будет обрабатывать доставку сигналов, он наследует текущую маску сигналов. Поскольку функция `sigwait` разблокирует сигналы, только один поток сможет получить их. Это позволяет при написании программы не беспокоиться о том, что главный поток может быть прерван этими сигналами.

Запустив эту программу, мы получим результаты, похожие на те, что дала нам программа из листинга 10.16:

```
$ ./a.out
^?                          введем символ прерывания
прерывание
^?                          еще раз введем символ прерывания
```



```
прерывание
^?                          и еще раз
прерывание
^\ $                        а теперь введем символ завершения
```

12.9. Потоки и fork