Лекция 3
Многопоточное программирование
POSIX Threads (2)

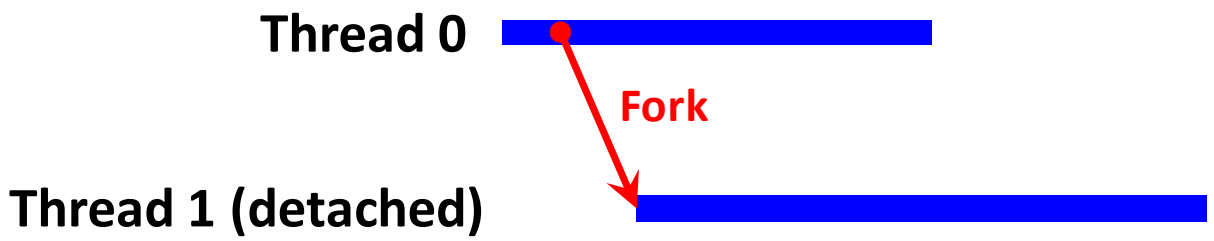
Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com
WWW: www.mkurnosov.net

Курс «Параллельные вычислительные технологии»
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)
Весенний семестр, 2017

Отсоединенные потоки (detached thread)

- **Отсоединений поток (detached thread)** – поток, дождаться завершения которого в другом потоке невозможно (при его завершении система автоматически освобождает ресурсы)
- По умолчанию, при создании, потоки являются присоединяемыми (joinable)
- Отсоединенный поток нельзя вновь сделать присоединяемым



- `int pthread_detach(pthread_t thread);`
- `pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);`

Создание отсоединенного потока

```
int main()
{
    printf("Master thread start: %f\n", wtime());
    pthread_attr_t attr;
    if (pthread_attr_init(&attr) != 0) {
        fprintf(stderr, "Can not init thread attributed\n"); exit(EXIT_FAILURE);
    }
    if (pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED) != 0) {  // Установка атрибута
        fprintf(stderr, "Can not set detach state\n"); exit(EXIT_FAILURE);
    }
    pthread_t tid;
    if (pthread_create(&tid, &attr, thread_func, NULL) != 0) {      // Запуск потока с заданными атрибутами
        fprintf(stderr, "Can't create thread\n"); exit(EXIT_FAILURE);
    }
    print_thread_state("Slave thread state", &attr);
    pthread_attr_destroy(&attr);

    sleep(3);
    printf("Master thread end: %f\n", wtime());
    pthread_exit(NULL);  // Завершаем мастер-поток, отсоединенные потоки продолжаются выполняться
    return 0;
}
```

Создание отсоединенного потока (продолжение)

```c
#include <unistd.h>     // for sleep
#include <sys/time.h>   // for gettimeofday
#include <pthread.h>

double wtime() {
    struct timeval t;
    gettimeofday(&t, NULL);
    return (double)t.tv_sec + (double)t.tv_usec * 1E-6;
}

void print_thread_state(const char *prefix, pthread_attr_t *attr)
{
    int detachstate;
    pthread_attr_getdetachstate(attr, &detachstate);
    printf("%s: thread is %s\n", prefix, (detachstate == PTHREAD_CREATE_DETACHED) ?
                                        "detached" : "joinable");
}

void *thread_func(void *arg)
{
    printf("Slave thread start: %f\n", wtime());
    sleep(10);
    printf("Slave thread end: %f\n", wtime());
    return NULL;
}
```

```
$ ./detach
Master thread start: 1487427230.027063
 Slave thread state: thread is detached
 Slave thread start: 1487427230.027308
  Master thread end: 1487427233.027775
   Slave thread end: 1487427240.027442
```

Перевод потока в состояние отсоединенного

```
int main()
{
    printf("Master thread start: %f\n", wtime());
    pthread_t tid1, tid2;

    if (pthread_create(&tid1, NULL, thread_func1, NULL) != 0) {
        fprintf(stderr, "Can't create thread\n");
        exit(EXIT_FAILURE);
    }

    if (pthread_create(&tid2, NULL, thread_func2, NULL) != 0) {
        fprintf(stderr, "Can't create thread\n");
        exit(EXIT_FAILURE);
    }
    pthread_detach(tid2);  // Мастер поток меняет состояние дочернего потока на DETACHED

    sleep(3);
    printf("Master thread end: %f\n", wtime());
    pthread_exit(NULL);
    return 0;
}
```

Перевод потока в состояние отсоединенного (продолжение)

```
void *thread_func1(void *arg)
{
    pthread_detach(pthread_self()); // Поток меняет свое состояние на отсоединенный

    printf("Slave1 thread start: %f\n", wtime());
    sleep(10);
    printf("Slave1 thread end: %f\n", wtime());
    return NULL;
}

void *thread_func2(void *arg)
{
    printf("Slave2 thread start: %f\n", wtime());
    sleep(10);
    printf("Slave2 thread end: %f\n", wtime());
    return NULL;
}
```

```
$ ./detach
Master thread start: 1487427816.134084
Slave1 thread start: 1487427816.134498
Slave2 thread start: 1487427816.134597
  Master thread end: 1487427819.134717
  Slave1 thread end: 1487427826.134621
  Slave2 thread end: 1487427826.134827
```

Многопоточная работа со связным списком

- Имеется связный список, заданный указателем на его голову (head)
- Два потока (writers) периодически добавляют новые элементы в голову списка
- Четыре потока (readers) проходя по списку и вычисляют число элементов в нем

Многопоточная работа со связным списком

```
struct listnode {
    int value;              // Data
    struct listnode *next;
};

struct listnode *list_createnode(int value)
{
    struct listnode *p;
    p = malloc(sizeof(*p));
    if (p != NULL) {
        p->value = value;
        p->next = NULL;
    }
    return p;
}

struct listnode *list_addfront(struct listnode *list, int value)
{
    struct listnode *newnode = list_createnode(value);
    if (newnode != NULL) {
        newnode->next = list;
        return newnode;
    }
    return list;
}
```

Многопоточная работа со связным списком

```
struct listnode *list_lookup(struct listnode *list, int value)
{
    for (; list != NULL; list = list->next) {
        if (list->value == value) {
            return list;
        }
    }
    return NULL;
}

int list_size(struct listnode *list)
{
    int size = 0;
    for (; list != NULL; list = list->next)
        size++;
    return size;
}
```

Многопоточная работа со связным списком

```
struct listnode *list_delete(struct listnode *list, int value)
{
    struct listnode *p, *prev = NULL;
    for (p = list; p != NULL; p = p->next) {
        if (p->value == value) {
            if (prev == NULL) {
                list = p->next;
            } else {
                prev->next = p->next;
            }
            free(p);
            return list;
        }
        prev = p;
    }
    return NULL;
}
```

Многопоточная работа со связным списком

```
int main()
{
    struct listnode *head = NULL; // Внимание: head – объект в стеке, указатель на него передан в потоки
    int nwriters = 2;
    pthread_t *writers = malloc(sizeof(*writers) * nwriters);
    for (int i = 0; i < nwriters; i++) {
        if (pthread_create(&writers[i], NULL, writer_thread, &head) != 0) {
            fprintf(stderr, "Can't create thread\n");
            exit(EXIT_FAILURE);
        }
    }
    int nreaders = 4;
    pthread_t *readers = malloc(sizeof(*readers) * nreaders);
    for (int i = 0; i < nreaders; i++) {
        if (pthread_create(&readers[i], NULL, reader_thread, &head) != 0) {
            fprintf(stderr, "Can't create thread\n");
            exit(EXIT_FAILURE);
        }
    }

    for (int i = 0; i < nwriters; i++) pthread_join(writers[i], NULL);
    for (int i = 0; i < nreaders; i++) pthread_join(readers[i], NULL);

    free(readers); free(writers);
    return 0;
}
```

Многопоточная работа со связным списком

```c
void *writer_thread(void *arg)
{
    struct listnode *head = (struct listnode *)arg;
    for (int i = 0; i < 10; i++) {
        head = list_addfront(head, i);
        printf("writer: add %d\n", i);
        sleep(1);
    }
    return NULL;
}

void *reader_thread(void *arg)
{
    struct listnode *head = (struct listnode *)arg;
    for (int i = 0; i < 10; i++) {
        int size = list_size(head);
        printf("reader: list size %d\n", size);
        sleep(1);
    }
    return NULL;
}
```

```
$ ./llist
reader: list size 4
reader: list size 4
reader: list size 4
reader: list size 4
writer: add 0
writer: add 0
reader: list size 4
reader: list size 4
reader: list size 4
reader: list size 4
reader: list size 4
writer: add 1
writer: add 1
...
```

Многопоточная работа со связным списком

```
void *writer_thread(void *arg)
{
    struct listnode *head = (struct listnode *)arg;
    for (int i = 0; i < 10; i++) {
        head = list_addfront(head, i);
        printf("writer: add %d\n", i);
        sleep(1);
    }
    return NULL;
}
```

Что произойдет если один поток добавляет элемент в список или уделает, а другой пытается выполнить проход по списку (lookup, size)?

```
    }
    return NULL;
}
```

```
$ ./llist
reader: list size 4
reader: list size 4
reader: list size 4
reader: list size 4
writer: add 0
writer: add 0
reader: list size 4
reader: list size 4
reader: list size 4
reader: list size 4
writer: add 1
writer: add 1
...
```

Многопоточная работа со связным списком

- Указатель на голову списка head – разделяемый ресурс
- Одновременная модификация связей между элементами списка (addfront, delete, deleteall) и их использование (lookup, size) может привести к некорректной работе
- Необходим синхронизация доступа к структуре списка

Многопоточная работа со связным списком

```
pthread_mutex_t llist_mutex = PTHREAD_MUTEX_INITIALIZER;     // Мьютекс синхронизации доступа к списку

void *writer_thread(void *arg)
{
    struct listnode *head = (struct listnode *)arg;
    for (int i = 0; i < 10; i++) {
        pthread_mutex_lock(&llist_mutex);
        head = list_addfront(head, i);
        pthread_mutex_unlock(&llist_mutex);
        printf("writer: add %d\n", i);
        sleep(1);
    }
    return NULL;
}

void *reader_thread(void *arg)
{
    struct listnode *head = (struct listnode *)arg;
    for (int i = 0; i < 10; i++) {
        pthread_mutex_lock(&llist_mutex);
        int size = list_size(head);
        pthread_mutex_unlock(&llist_mutex);
        printf("reader: list size %d\n", size);
        sleep(1);
    }
    return NULL;
}
```

- Критическая секция должна быть небольшой для минимизации времени последовательного выполнения кода (гранулярность, зернистость блокировки)
- Возможно блокировать лишь часть кода функций addfront и size

Многопоточная работа со связным списком: rwlock

```c
#define __USE_XOPEN2K   // for rwlock
#include <pthread.h>

pthread_rwlock_t llist_rwlock = PTHREAD_RWLOCK_INITIALIZER;

void *writer_thread(void *arg)
{
    struct listnode *head = (struct listnode *)arg;
    for (int i = 0; i < 10; i++) {
        pthread_rwlock_wrlock(&llist_rwlock);
        head = list_addfront(head, i);
        pthread_rwlock_unlock(&llist_rwlock);
    }
    return NULL;
}

void *reader_thread(void *arg)
{
    struct listnode *head = (struct listnode *)arg;
    for (int i = 0; i < 10; i++) {
        pthread_rwlock_rdlock(&llist_rwlock);
        int size = list_size(head);
        pthread_rwlock_unlock(&llist_rwlock);
    }
    return NULL;
}
```

- **rwlock** – блокировка (мьютекс) типа «чтение-запись»
- Если блокировка на запись свободна (wrlock), то блокировку на чтение (rdlock) захватить может произвольное число процессов
- Позволяет сократить накладные расходы на синхронизацию для приложений выполняющих чтение и модификацию разделяемых данных

Многопоточный счетчик

- Имеется глобальная переменная counter
- Потоки одновременно выполняют ее инкремент (counter++)

Многопоточный счетчик: mutex

```
int main(int argc, char **argv)
{
    double t = wtime();
    int counter = 0;

    int nthreads = argc > 1 ? atoi(argv[1]) : 16;
    pthread_t *tids = malloc(sizeof(*tids) * nthreads);
    if (tids == NULL) {
        fprintf(stderr, "No enough memory\n");
        exit(EXIT_FAILURE);
    }
    for (int i = 0; i < nthreads; i++) {
        if (pthread_create(&tids[i], NULL, counter_thread, (void *)&counter) != 0) {
            fprintf(stderr, "Can't create thread\n");
            exit(EXIT_FAILURE);
        }
    }

    for (int i = 0; i < nthreads; i++)
        pthread_join(tids[i], NULL);
    t = wtime() - t;
    printf("Counter (threads %d, counter %d): %.6f sec.\n", nthreads, counter, t);

    free(tids);
    return 0;
}
```

Многопоточный счетчик: mutex

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *counter_thread(void *counter)
{
    for (int i = 0; i < 5000000; i++) {
        pthread_mutex_lock(&mutex);
        (*((int *)counter))++;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

```
$ ./counter
Counter (threads 16, counter 80000000): 10.733569 sec.

$ ./counter
Counter (threads 16, counter 80000000): 9.001701 sec.
```

- Если мьютекс уже захвачен, операционная система отправляет поток в состояние ожидания (снимается с выполнения)
- При освобождении мьютекса потоки, ранее пытавшиеся его захватить, пробуждаются и пытаются вновь его захватить
- Смена состояния потока ресурсоемкая операция
- Мьютексами желательно защищать участки кода, время выполнения которых больше времени перевода потока в состояние ожидания и последующего его «пробуждения»

Многопоточный счетчик: spinlock

```c
#define __USE_XOPEN2K   // for spinlock
#include <pthread.h>

pthread_spinlock_t spinlock;

void *counter_thread(void *counter)
{
    for (int i = 0; i < 5000000; i++) {
        pthread_spin_lock(&spinlock);
        (*((int *)counter))++;
        pthread_spin_unlock(&spinlock);
    }
    return NULL;
}
```

- Если spinlock уже захвачен, поток запускает цикл активного ожидания освобождения блокировки (в пространстве пользователя);
- Поток не меняет своего состояния и не снимается с процессора
- Спинлоками желательно защищать участки кода, время выполнения которых незначительно

Многопоточный счетчик: spinlock

```
int main(int argc, char **argv)
{
    double t = wtime();
    pthread_spin_init(&spinlock, PTHREAD_PROCESS_PRIVATE);
    int counter = 0;

    int nthreads = argc > 1 ? atoi(argv[1]) : 16;
    pthread_t *tids = malloc(sizeof(*tids) * nthreads);
    for (int i = 0; i < nthreads; i++) {
        if (pthread_create(&tids[i], NULL, counter_thread, (void *)&counter) != 0) {
            fprintf(stderr, "Can't create thread\n");
            exit(EXIT_FAILURE);
        }
    }
    for (int i = 0; i < nthreads; i++)
        pthread_join(tids[i], NULL);
    t = wtime() - t;
    printf("Counter (threads %d, counter %d): %.6f sec.\n", nthreads, counter, t);

    pthread_spin_destroy(&spinlock);
    free(tids);
    return 0;
}
```

```
$ ./counter
Counter (threads 16, counter 80000000): 7.400520 sec.
```