Средства и системы параллельного программирования

Семинар #2
Синхронизация потоков в pthreads

Почему не всё так плохо?

У нас есть две ключевые гарантии, предоставляемые аппаратно:

1. **Write propagation (распространение записи)**
   Никакое ядро не сможет бесконечно читать старое значение переменной, если оно обновлено потоком на другом ядре.

2. **Transaction serialization (единый порядок транзакций)**
   Для **каждого отдельного адреса** существует единый порядок операций чтения/записи, **одинаково видимый всеми ядрами**.

Что нас не устраивает?

1. Нам часто нужен определённый порядок выполнения инструкций (доступы к ресурсам, и. т.п)
2. Многие операции не атомарны, например a += value;

Базовый подход - Peterson algorithm

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
    int other;                          /* number of the other process */

    other = 1 – process;                /* the opposite of process */
    interested[process] = TRUE;         /* show that you are interested */
    turn = process;                     /* set flag */
    while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
    interested[process] = FALSE;        /* indicate departure from critical region */
}
```

в чём недостаток?

Test and set lock (TSL)

TSL RX, LOCK - атомарная запись на регистр значения слова LOCK, присваивание LOCK ненулевого значения.

Основной момент - процессор, выполняющий TSL, блокирует шину памяти ВСЕМ ядрам!

```
int test_and_set(int* lockPtr) {
    int oldValue;
    // -- Start of atomic segment --
    oldValue = *lockPtr;
    *lockPtr = LOCKED;
    // -- End of atomic segment --
    return oldValue;
}
```

Test and set lock (TSL) (2)

```
enter region:
    TSL REGISTER,LOCK
    CMP REGISTER,#0
    JNE enter_region

    RET


leave region:
    MOVE LOCK,#0
    RET
```

Особого выигрыша от использования TSL не получаем

Spin-блокировка

Spin-блокировка - активная блокировка, при которой поток не переходит в системное состояние ожидания

```
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
```

lock - указатель на переменную спин-лока, pshared - видимость спин-лока, PTHREAD_PROCESS_SHARED (межпроцессное взаимодействие) или PTHREAD_PROCESS_PRIVATE (только в рамках одного процесса)

```
int pthread_spin_destroy(pthread_spinlock_t *lock);
```

Разрушение спин-блокировки по адресу lock

Для того, чтобы блокировка была верной, все потоки должны видеть одну и ту же переменную типа pthread_spinlock_t (лучше держать её в глобальных переменных)

Spin-блокировка

```
int pthread_spin_lock(pthread_spinlock_t *lock);
```

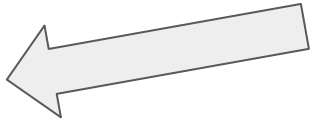
Захват блокировки по адресу lock. Если поток захватил блокировку, идём выполнять следующую команду. Нет - крутимся в активном ожидании

```
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

После выполнения нужных операций с разделяемым ресурсом, необходимо освободить блокировку

Mutex

```
mutex_lock:
    TSL REGISTER,MUTEX

    CMP REGISTER,#0
    JZE ok


    CALL thread_yield

    JMP mutex lock
ok:    RET



mutex_unlock:
    MOVE MUTEX,#0
    RET
```



Mutex (2)

```
int pthread_mutex_init(pthread_mutex_t *mutex,
pthread_mutexattr_t *attr);
```

Инициализация переменной-мьютекса по адресу mutex

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Освобождение переменной mutex

Для того, чтобы блокировка была верной, все потоки должны видеть одну и ту же переменную типа pthread_mutex_t (лучше держать её в глобальных переменных)

pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER; - дефолтный мьютекс можно инициализировать и вот так, init не нужен

Атрибуты мьютекса

```
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);

int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);
```

PTHREAD_MUTEX_NORMAL - при попытке повторного захвата мьютекса (без разблокировки) будет дэдлок

PTHREAD_MUTEX_ERRORCHECK - при попытке повторного захвата мьютекса (без разблокировки) выдаст ошибку

PTHREAD_MUTEX_RECURSIVE - можно повторно захватывать несколько раз (освободить нужно столько же раз)

PTHREAD_PROCESS_SHARED или PTHREAD_PROCESS_PRIVATE - ставим возможность другим ПРОЦЕССАМ брать мьютекс (по умолчанию PTHREAD_PROCESS_PRIVATE)

Mutex (2)

int pthread_mutex_lock(pthread_mutex_t *mutex);

Блокирующий захват мьютекса. Ждёт, пока мьютекс mutex станет свободным, и делает текущий поток владельцем. Если мьютекс ждут несколько потоков, **новый владелец определяется СЛУЧАЙНО (не обязательно тот, кто пришёл первый)**

int pthread_mutex_unlock(pthread_mutex_t *mutex);

Освобождение мьютекса mutex. Разрешено только потоку-владельцу.

int pthread_mutex_trylock(pthread_mutex_t *mutex);

Неблокирующий захват. Возврат: 0 — успех (захватили мьютекс). EBUSY — занято

Barrier

```
int pthread_barrier_init(pthread_barrier_t  *barrier,  const
pthread_barrierattr_t *restrict attr, unsigned count);
```

Инициализирует барьер на count потоков (count > 0)

```
int pthread_barrier_wait(pthread_barrier_t  *barrier);
```

Задаёт точку синхронизации между потоками

Функция возвращает управление в программу, когда count потоков дошли до этой точки. Одному из потоков возвращает *PTHREAD_BARRIER_SERIAL_THREAD*, остальным 0.

Барьер циклический — после срабатывания им можно пользоваться снова для следующей "фазы"

```
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

Освобождение переменной barrier

Типичная ситуация

```
lock(mutex);
while(!flag) {
   sleep(100);
}
unlock(mutex);
```

Conditional variables (0)

pthread_cond_t

pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);

Инициализирует условную переменную по адресу cond, передавая ей атрибуты attr

pthread_cond_destroy(pthread_cond_t *cond);

Разрушает условную переменную по адресу cond и освобождает связанные ресурсы

Conditional variables (1)

pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);

атомарно: освободить удерживаемый мьютекс m и положить себя (поток) в очередь ожидания сигнала.

pthread_cond_signal(pthread_cond_t *c);

Просигнализировать другому потоку из очереди ожидания (**NB: очередь не FIFO, по сигналу пробудится случайный спящий на c поток!. То есть правильнее называть очередь множеством**), что теперь можно перейти в состояние борьбы за мьютекс. Пробуждаемый поток должен был до этого заснуть тоже на c. Если таких потоков нет, сигнал утерян!

pthread_cond_broadcast (pthread_cond_t *c);

То же самое, что и signal, только сигнал идёт всем спящим на c потокам. Все спящие потоки переходят в состояние борьбы за мьютекс

Conditional variables (2) - а можно без мьютексов?

```
void child_func() {
    done = 1;
    pthread_cond_signal(&c);
}
void parent_func() {
    if (done == 0) {
        pthread_cond_wait(&c);
    }
}
```

Conditional variables (3) - а можно без state?

```
void child_func() {
     pthread_mutex_lock(&m);
     pthread_cond_signal(&c);
     pthread_mutex_unlock(&m);
}

void parent_func() {
     pthread_mutex_lock(&m);
     pthread_cond_wait(&c, &m);
     pthread_mutex_unlock(&m);
}
```

Conditional variable (4)

Всегда (99%) следует работать с conditional_variable под мьютексом и с проверкой какого-то предиката (состояния, размера очереди, флага готовности и.т.п) !

Для проверки предиката всегда (100%) используем while-цикл, а не if! (Spurious wakeups & Mesa semantics)

Задание

(Разрешается использование C++, для организации многопоточности использовать pthreads)

Реализовать класс/структуру с семантикой многопоточной очереди со следующими свойствами:

- атрибут **queue** - произвольная ("*однопоточная*") реализация очереди фиксированного размера. Тип данных - любой POD-тип.
- метод **put**() - положить один элемент в конец очереди
- метод **get**() - взять один (первый) элемент из очереди
- методы могут вызываться одновременно различным (произвольным) числом потоков
- **put**() и **get**() должны корректно работать при любом состоянии очереди - при пустой очереди **get**() - поток ждёт появления элемента, при полной очереди **put**()-поток ждёт освобождения места в очереди
- Если поток A сделал put() или get() раньше потока B, не обязательно, что A выйдет из метода раньше B

Разрешается добавлять дополнительные атрибуты в класс

Снабдить решение множеством тестов ({1 пишущий поток, N читающих}, {N, 1}, {M, N}, {1, 1}). Программа должна корректно работать при различных конфигурациях M и N.

Дедлайн: 29.09, 6.10. Также вместе с заданием необходимо прислать публичный ключ!