Синхронизация и межпроцессное взаимодействие, ч. 1

Мультипроцессорные среды

- В современных процессорах много ядер
- Каждое ядро может исполнять код независимо от других
- Код не всегда исполняется последовательно (например, спекулятивное исполнение)
- Разные ядра могут по-разному видеть порядок операций

Модель памяти

- Модель памяти накладывает ограничения на то, как процессор может переставлять операции местами
- Модель памяти x86 описывается в ISDM (Vol. 3A, Chapter 9)
- `std::memory_order` – другая модель памяти C++ (более общая, более сложная)
- Модели памяти не являются предметом нашего курса :(

```
x = 0, y = 0
```

```
; core 1
mov [x], 1
mov [y], 1
```

```
; core 2
mov r1, [y]
mov r2, [x]
```

Какие варианты возможны?

1. `r1 = 0` `r2 = 0`
2. `r1 = 1` `r2 = 1`
3. `r1 = 0` `r2 = 1`
4. `r1 = 1` `r2 = 0`

Модель памяти: только 4 вариант невоможен!

Атомарность

- Атомарные операции – операции, которые не могут наблюдаться другими ядрами посреди исполнения
- Атомарная операция либо ещё не выполнена, либо уже выполнена
- Выровненная запись или чтение – атомарные операции
- `add [rsp], rax` – неатомарная операция, она «развернётся» в:

```
mov internal_reg, [rsp]
add internal_reg, rax
mov [rsp], internal_reg
```

- Read-modify-write
- Неатомарные операции требуют *синхронизации*

Процессы > ядра

- В современных ОС (runnable) процессов бывает значительно больше, чем ядер
- Процессы образуют *очередь на выполнение*
- Какой процесс будет выполнен следующим из очереди – решает *планировщик* (scheduler)

Многозадачность

Кооперативная

- Процессы добровольно передают друг другу управление
- Если один из процессов зависнет – все остальные процессы будут бесконечно ждать его

Вытесняющая

- Операционная система сама вытесняет процессы раз в несколько миллисекунд
- Процессы также могут самостоятельно отдавать управление
- Выделяемый квант времени может варьироваться
- Прерывание рабочего процесса – затратная операция

Переключение контекста

- Процесс переключающий процесс на текущем ядре называется *переключением контекста* (или *context switch*)
- Это операцию делает код внутри ядра
- Context switch довольно затратен, но необходим для многозадачности
- Современные процессоры могут эффективно выполнять сотни CS на одном ядре в секунду

Пример: снятие денег с банковского счёта

```
void withdraw(bank_acc_t* acc, int amount) {
    if (acc->balance < amount) {
        return;
    }
    acc->balance -= amount;
    // ...
}

int main() {
    acc->balance = 100;

    thread1 { withdraw(acc, 80); }

    thread2 { withdraw(acc, 90); }

    // acc->balance == 20 или acc->balance == 10?
}
```

Race condition и data race

- Race condition – ситуация, когда результат её выполнения зависит от последовательности выполнения потоков и операций в них
- Data race – ситуация, когда два потока пишут что-то в память, не используя никакую синхронизацию

Compare-and-exchange

- Compare-and-exchange (compare-and-swap, CAS) – атомарная операция в процессоре
- `cmpxchg` в x86
- `ldrex` / `strex` в ARM

```
_Bool atomic_cas(A* obj, C* expected, C desired) {
    // Псевдокод!
    atomic {
        if (*obj == *expected) {
            *obj = desired;
            return true
        }
        *expected = *obj;
        return false;
    }
}
```

Spinlock

```
typedef atomic_int spinlock_t;

void spin_lock(spinlock_t* lock) {
    while (true) {
        int expected = 0;
        if (atomic_cas(lock, &expected, 1)) {
            break;
        }
    }
}

void spin_unlock(spinlock_t* lock) {
    atomic_store(lock, 0);
}
```

Исправленный пример

```
void withdraw(bank_acc_t* acc, int amount) {
    spin_lock(&acc->lock);

    int b = acc->balance;
    b += amount;
    acc->balance = b;

    spin_unlock(&acc->lock);
    // ...
}

int main() {
    acc->balance = 100;

    thread1 { withdraw(acc, 80); }
    thread2 { withdraw(acc, 90); }

    // acc->balance == 20 или acc->balance == 10
}
```

Спинлоки: проблемы

- Процессы потребляют CPU во время ожидания

Спинлоки: проблемы

- Процессы потребляют CPU во время ожидания
- Инверсия приоритетов (приводит к deadlock)

```
void withdraw(bank_acc_t* acc, int amount) {
    spin_lock(&acc->lock);

    int b = acc->balance;
    b += amount;
    acc->balance = b;

    spin_unlock(&acc->lock);
    // ...
}

int main() {
    acc->balance = 100;

    thread1 { withdraw(acc, 80); }
    on_signal { withdraw(acc, 90); }
}
```

Спинлоки: проблемы

- Процессы потребляют CPU во время ожидания
- Инверсия приоритетов (приводит к deadlock)
- Если локи берутся в разном порядке, то тоже может возникать deadlock (dining philosophers problem)

```
void transfer(bank_acc_t* a, bank_acc_t* b, int amount) {
    spin_lock(&a->lock);
    spin_lock(&b->lock);

    a->balance -= amount;
    b->balance += amount;

    spin_unlock(&b->lock);
    spin_unlock(&a->lock);
}

int main() {
    thread1 { transfer(x, y, 1000) }
    thread2 { transfer(y, x, 1000) }

    // ???
}
```

Mutex

- Mutex = mutual exclusion
- «Засыпающий спинлок»: вместо бесконечного цикла – переводит процесс в состояние сна
- При разблокировке – отпустить лок и разбудить все ждущие процессы
- `man 7 futex`

```c
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t* mutex);
int pthread_mutex_unlock(pthread_mutex_t* mutex);
```

`pthread_mutex`: optimistic locking

- Если мьютекс занят, то с большой вероятностью он довольно скоро освободится
- Переход в контекст ядра – дорого
- Поэтому mutex проворачивает несколько сотен итераций и только потом засыпает по-настоящему

Read-write lock

- Обычный мьютекс позволяет изменять структуру только одному потоку, но и *читать* можно только одному потоку
- Разделим взятия лока на чтения, и на записи
- Сколько угодно чтений может быть параллельно, запись не может пересекаться с чтением или другой записью
- Вуаля, читатели не блокируют друг друга!

```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t* rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t* rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t* rwlock);
```

Read-write lock: writer starvation

- `rwlock` потенциально может оказаться в ситуации, когда читатели приходят и приходят, а писатель ждёт своей очереди
- Это состояние называется *writer starvation*

```
int pthread_rwlockattr_setkind_np(
    pthread_rwlockattr_t* attr,
    int pref);

int pthread_rwlock_init(
    pthread_rwlock_t* rwlock,
    const pthread_rwlockattr_t* attr);
```

Пример: очередь задач

```
void enqueue(queue_t* q, fn_t fn) {
    pthread_mutex_lock(&q->mutex);
    queue_push(q, fn);
    pthread_mutex_unlock(&q->mutex);
}

fn_t dequeue(queue_t* q) {
    // wait until q is non-empty — ?

    pthread_mutex_lock(&q->mutex);
    fn = queue_pop(q);
    pthread_mutex_unlock(&q->mutex);
    return fn;
}

int main() {
    for (int i = 0; i < 10; i++) {
        thread {
            while (true) {
                fn = dequeue();
                fn();
            }
        }
    }
}
```

```
fn_t dequeue(queue_t* q) {
    if (queue_empty(q)) {
        fall_sleep();
    }

    pthread_mutex_lock(&q->mutex);
    fn = queue_pop(q);
    pthread_mutex_unlock(&q->mutex);
    return fn;
}

void enqueue(queue_t* q, fn_t fn) {
    pthread_mutex_lock(&q->mutex);
    queue_push(q, fn);
    pthread_mutex_unlock(&q->mutex);
    wake_all_sleeping_threads();
}
```

Missing wakeup problem

```
T1: if (queue_empty(q)) { // true
T2: pthread_mutex_lock(&q->mutex);
T2: queue_push(q, fn);
T2: pthread_mutex_unlock(&q->mutex);
T2: wake_all_sleeping_threads();
T1: fall_sleep();
```

T1 навсегда заблокируется, хотя в очереди есть задачи для обработки!

```
fn_t dequeue(queue_t* q) {
    if (queue_empty(q)) {
        fall_sleep();
    }

    pthread_mutex_lock(&q->mutex);
    fn = queue_pop(q);
    pthread_mutex_unlock(&q->mutex);
    return fn;
}

void enqueue(queue_t* q, fn_t fn) {
    pthread_mutex_lock(&q->mutex);
    queue_push(q, fn);
    pthread_mutex_unlock(&q->mutex);
    wake_all_sleeping_threads();
}
```

Чиним missing wake up problem

```
fn_t dequeue(queue_t* q) {
    pthread_mutex_lock(&q->mutex);
    if (queue_empty(q)) {
        // атомарно отпустит мьютекс и переведёт поток в сон
        sleep_until_wakeup(&q->mutex);
        // возьмёт мьютекс при выходе
    }
    fn = queue_pop(q);
    pthread_mutex_unlock(&q->mutex);
    return fn;
}

void enqueue(queue_t* q, fn_t fn) {
    pthread_mutex_lock(&q->mutex);
    queue_push(q, fn);
    wake_all_sleeping_threads();
    pthread_mutex_unlock(&q->mutex);
}
```

Spurious wakeup

- Два `dequeue` могут быть разбужены конкурентно, что приведёт к `queue_pop` из пустой очереди
- Сам `sleep_until_wakeup` может «ложно» (spurious) выйти
- Для починки нужно добавить `while`

```
fn_t dequeue(queue_t* q) {
    pthread_mutex_lock(&q->mutex);
    while (queue_empty(q)) {
        sleep_until_wakeup(&q->mutex);
    }
    fn = queue_pop(q);
    pthread_mutex_unlock(&q->mutex);
    return fn;
}

void enqueue(queue_t* q, fn_t fn) {
    pthread_mutex_lock(&q->mutex);
    queue_push(q, fn);
    wake_all_sleeping_threads();
    pthread_mutex_unlock(&q->mutex);
}
```

pthread_cond_t

```c
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mutex);
int pthread_cond_broadcast(pthread_cond_t* cond);
int pthread_cond_signal(pthread_cond_t* cond);
```

```c
fn_t dequeue(queue_t* q) {
    pthread_mutex_lock(&q->mutex);
    while (queue_empty(q)) {
        pthread_cond_wait(&q->condvar, &q->mutex);
    }
    fn = queue_pop(q);
    pthread_mutex_unlock(&q->mutex);
    return fn;
}

void enqueue(queue_t* q, fn_t fn) {
    pthread_mutex_lock(&q->mutex);
    queue_push(q, fn);
    pthread_mutex_unlock(&q->mutex);
    pthread_cond_broadcast(&q->cond);
}
```

Mutex: минусы

- Процессы и мьютексы не «дружат» между собой
- Если потоко зависнет с залоченным мьютексом (например, major page fault), все остальные потоки его будут ждать
- Если процесс будет убит во время критической секции, все остальные процессы зависнут на взятии мьютекса

```
void process1() {
    pthread_mutex_lock(&mut);
    // Убит ОС!
    pthread_mutex_unlock(&mut);
}

void process2() {
    pthread_mutex_lock(&mut);
    // ...
    pthread_mutex_unlock(&mut);
}
```

Lock-free алгоритмы

Интуитивное определение – алгоритмы синхронизации, которые *не используют локи*

Lock-free стек

- CAS используется в качестве примитива синхронизации
- Если цикл провернулся (CAS вернул false) – какой-то другой поток поменял вершину стека

```
typedef struct {
    struct node* next;
    int x;
} node_t;

typedef struct {
    node_t* top;
} stack_t;
```

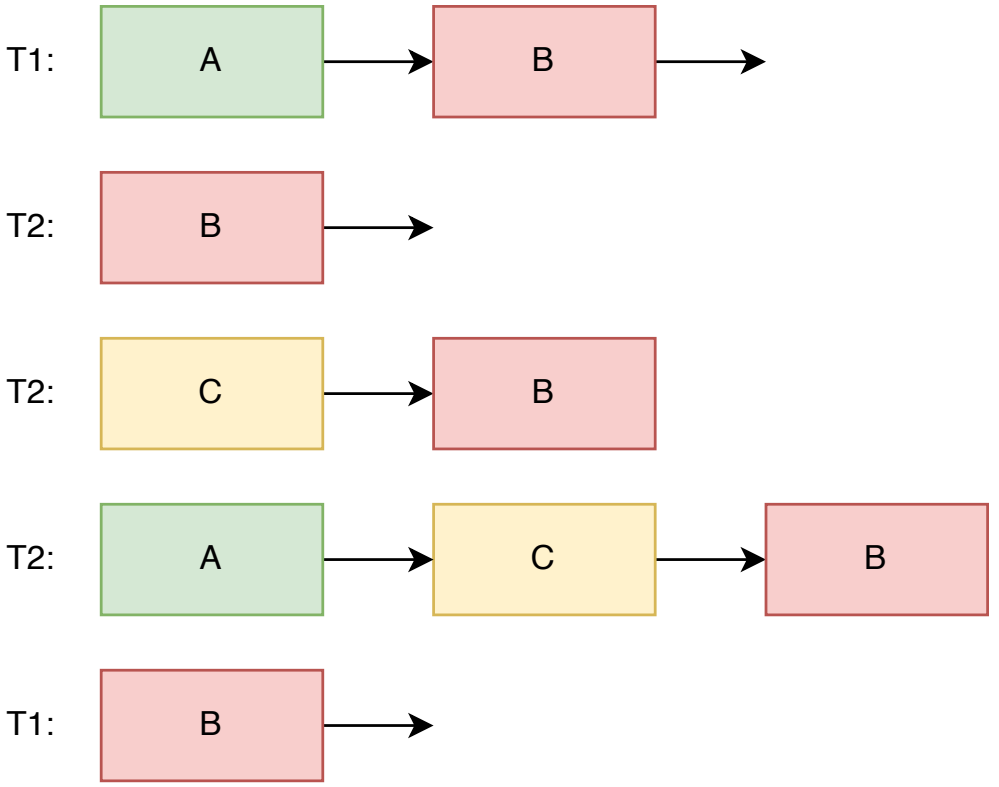
```
void push(stack_t* st, node_t* n) {
    while (true) {
        node_t* old_top = st->top;
        n->next = old_top;
        if (atomic_cas(&st->top, &old_top, n)) {
            break;
        }
    }
}

void pop(stack_t* st, node_t* n) {
    node_t* top = st->top;
    while (top != NULL) {
        if (atomic_cas(&st->top, &top, top->next)) {
            break;
        }
    }
    // ...
}
```

Lock-free: ABA problem

```
T1: начинает pop()
T1: top = st->top; // A
T1: top_next = B
T2: pop() = A
T2: push(C)
T2: push(A)
T1: cas(&st->top, &top, top_next) // cas(A, A, B)
T1: C потерялся!
```

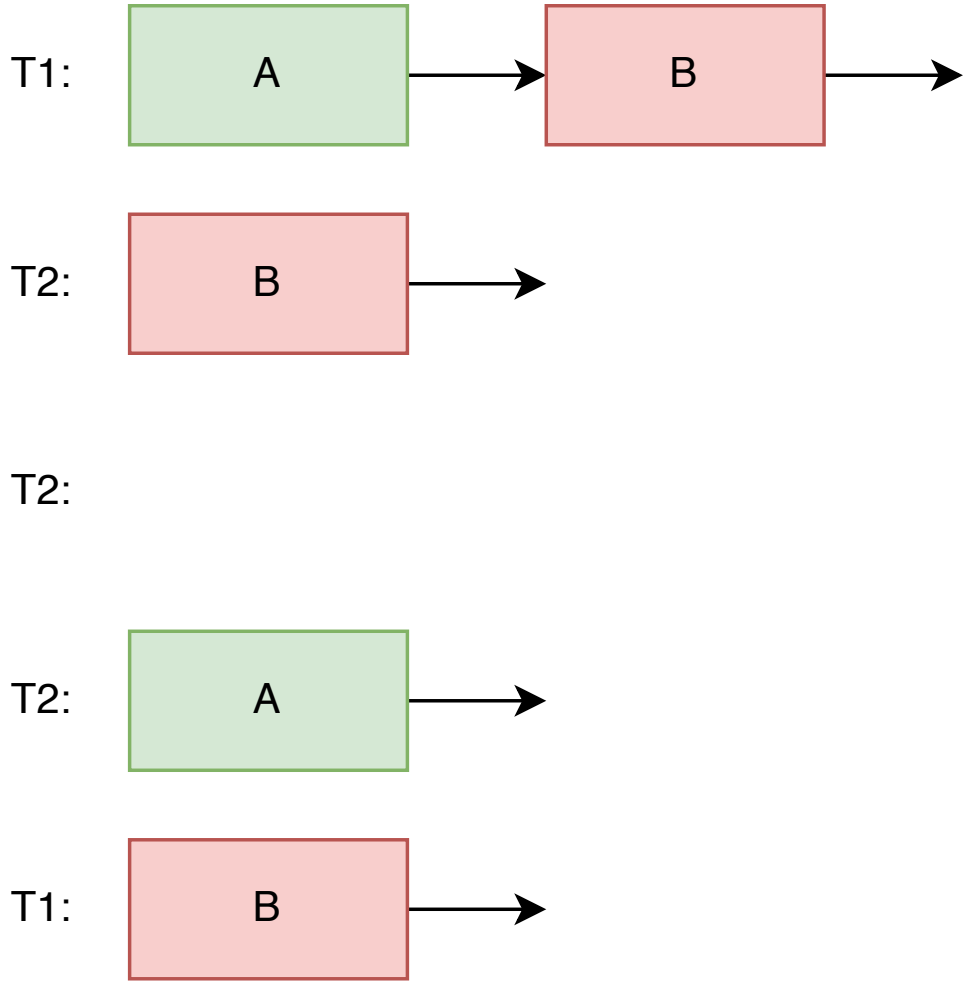
CAS (в нашем случае) сравнивает *адреса*, но не элементы!



ABA проблема и освобождение памяти

```
void pop(stack_t* st, node_t* n) {
    node_t* top = st->top;
    while (top != NULL) {
        if (atomic_cas(&st->top, &top, top->next)) {
            break;
        }
    }
    free(n); // <----
}
```

```
T1: top = st->top; // A
T1: top_next = B
T2: pop() = A
T2: free(A)
T2: pop() = B
T2: free(B)
T2: push(A)
T1: cas(&st->top, &top, top_next) // cas(A, A, B)
T1: st->top == B
```



ABA проблема: tagged pointers

- Нижние биты указателей (3 бита), возвращаемые `malloc`, будут нулями, т.к. память выровненная
- Эти биты можно использовать как версии указателя и тогда `cas` не пройдёт, т.к у A будет другой тэг
- Всего 3 бита = 8 значений, может быть переполнение и ABA опять вернётся :(

ABA проблема: double word CAS и другие варианты

- На некоторых платформах возможно реализовать CAS, который делает обновление по двум машинным словам
- В стеке можно завести постоянно увеличивающийся счётчик
- По-сути – аналог tagged pointers, но с большим количеством тэгов
- Intermediate nodes
- Hazard pointers

Минусы lock-free

- Lock-free гарантирует, что система *в целом делает прогресс*, однако допускают ситуацию, когда один поток выполняется *вечно*
- Lock-free алгоритмы сложнее, чем аналоги на блокировках (ABA проблема)
- Освобождение памяти не такое простое

Þakka þér fyrir!