Механизмы синхронизации

A. Циклический барьер - 5 баллов

Барьер (barrier) - примитив синхронизации для n потоков, который представляет собой виртуальную границу в коде, пересечь которую поток может только в случае, если к ней подошли все n потоков.

Барьер реализует единственный метод: void pass()

Когда поток хочет пройти через барьер, он вызывает этот метод и блокируется до тех пор, пока к барьеру не подойдут все остальные потоки (т.е. до тех пор, пока все они тоже не вызовут pass()). Поток, последним вызвавший pass(), разблокирует все ожидающие потоки, после чего барьер считается пройденным.

При создании барьер параметризуется числом потоков, которые будут проходить через него.

Барьер называется **циклическим** (cyclic), если через него можно проходить многократно. Т.е. поток, прошедший через метод pass, может вызвать его повторно, и снова ждать всех остальных потоков.

Пример практического применения барьеров - параллельные итеративные алгоритмы, в которых работа на каждой итерации разбивается по данным на независимые части и выполняется в отдельных потоках, и перейти к следующей итерации можно только если все потоки уже завершили работу на предыдущей итерации.

Пример таких задач: вычисление префиксных сумм или bottom-up FFT (быстрое преобразование Фурье)

Реализуйте циклический барьер с помощью условных переменных или семафоров.

Подсказка:

Сложность реализации циклического барьера состоит в обработке следующей ситуации: поток, только покинувший барьер, может тут же

подойти к нему снова, пока остальные потоки его еще не покинули. Поэтому идея с простым счетчиком потоков не работает.

B. Шагающий робот - 2+2+1 балла

У робота есть две ноги, движением каждой управляет отдельный поток. Потоки могут выполнять только одно движение - шаг. Для того, чтобы робот мог нормально ходить, потоки должны выполнять шаги строго по очереди:

Поток 1	**Поток 2**
step(“left”)	
	step(“right”)
step(“left”)	
	step(“right”)

Для определенности будем считать, что робот делает первый шаг левой ногой.

Придумайте механизм координации потоков с помощью:

1) Условных переменных

2) Семафоров + реализуйте семафор с помощью условных переменных

3) Используя семафоры, реализуйте робота-многоножку с n ногами, которыми он должен перебирать по циклу.

C. Адаптивный мьютекс с помощью фьютекса - 5 баллов

Пусть мы хотим реализовать идеальный мьютекс. Какими свойствами он должен обладать?

1) Захват свободного мьютекса, на владение которым никто больше не претендует, должен происходить очень быстро, без погружения в ядро операционной системы.

Посмотрим на TAS спинлок. Если спинлок свободен, и есть только один поток, который претендует на него, то этот поток выполнит одну операцию test-and-set и захватит спинлок. Один щелчок атомарной операцией, выполнение которой занимает единицы тактов процессора - и мьютекс захвачен.

2) Если поток не может захватить мьютекс, потому что его опередил другой поток (такую ситуацию называют **contention**, т.е. соревнованием за мьютекс), то этот поток должен заблокироваться внутри ядра операционной системы до тех пор, пока мьютекс не освободится.

TAS спинлок так делать не умеет: он будет крутиться в цикле ожидания и греть процессор до тех пор, пока не захватит флаг locked с помощью операции test-and-set.

Первый сценарий называется **fast path**. Если мьютекс умеет блокироваться в ядре операционной системы, но при этом имеет fast path, то такой мьютекс называется **адаптивным**.

Для реализации адаптивного мьютекса операционная система предлагает инструмент - **фьютекс**.

Фьютекс – это список (очередь) заблокированных потоков в ядре операционной системы, привязанная к ячейке памяти программы.

Фьютекс поддерживает две операции:

- **futex-wait**(addr, expected) – если содержимое ячейки addr == expected, то остановить исполнение текущего потока и добавить его в очередь ожидания, иначе выйти из вызова

- **futex-wake**(addr, k) – разбудить k потоков в очереди, связанной с ячейкой addr.

Семантика значения в ячейке addr фьютексу неизвестна.

В ядре эти операции исполняются под спинлоком ⇒ вызов **futex-wake** не может выполниться между проверкой значения addr и остановкой потока внутри вызова **futex-wait**.

Рассмотрим следующую реализацию адаптивного мьютекса с помощью фьютекса:

```
void adaptive_mutex::init() {
    locked = 0
}

void adaptive_mutex::lock() {
    while (test-and-set(locked)) {
        futex-wait(locked, 1)
    }
}

void adaptive_mutex::unlock() {
    locked = 0
    futex-wake(locked, 1)
}
```

У этой реализации есть недостаток: каждый unlock требует системного вызова futex-wake, даже если поток всего один.

Предложите реализацию, в которой поток в unlock не будет вызывать futex-wake в uncontended случае, т.е. в случае, когда нет других потоков, которые претендовали бы на мьютекс.

Указание: Вместо двух состояний (0 - свободен, 1 - захвачен) нужно использовать три: 0 - свободен, 1 - захвачен, но других потоков нет,

а значит будить в unlock никого не надо, 2 - захвачен, но есть потоки, которые претендуют на захват и могут уснуть в futex-wait.
