Третья работа

Семафор

Семафор – механизм синхронизации доступа к критическим ресурсам

Семафор инициализируется **ненулевым** значением

```
/* A counting semaphore. */
struct semaphore
  {
    unsigned value;             /* Current value. */
    struct list waiters;        /* List of waiting threads. */
  };
```

Операции над семафором

Операция "Down" или "P", которая ожидает, когда переменная семафора станет положительной, а затем декрементирует ее

```
void sema_down (struct semaphore *sema)
{
  while (sema->value == 0)
    {
      list_push_back (&sema->waiters, &thread_current ()->elem);
      thread_block ();
    }
  sema->value--;
}
```

Операция "Up" или "V", которая инкрементирует переменную семафора и пробуждает один ожидающий процесс, если такой существует

```
void sema_up (struct semaphore *sema)
{
  ASSERT (sema != NULL);
  if (!list_empty (&sema->waiters))
    thread_unblock (list_entry (list_pop_front (&sema->waiters),
                                struct thread, elem));
  sema->value++;
}
```

Down:
Блокировать процесс Пока Значение_Семафора == 0
Значение_Семафора -= 1

Up:
Разбудить ожидающий процесс
Значение_Семафора += 1

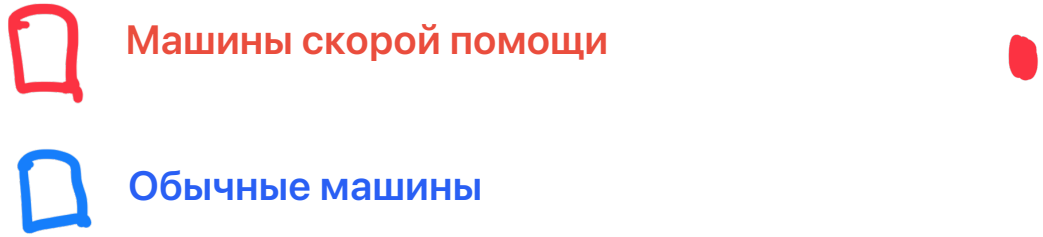
Примеры использования семафора Синхронизация потоков

```
import threading

semaphore = threading.Semaphore(0) # Инициализируем семафор с нулевым начальным значением

def thread1():
    # Выполнение некоторой критической работы
    semaphore.release() # Увеличиваем значение семафора, разрешая второму потоку продолжить

def thread2():
    print("Поток 2 ожидает")
    semaphore.acquire() # Ожидаем, пока первый поток разрешит доступ
    print("Поток 2 продолжает выполнение")

t1 = threading.Thread(target=thread1)
t2 = threading.Thread(target=thread2)
```

Примеры использования семафора Ограничение доступа к ресурсам

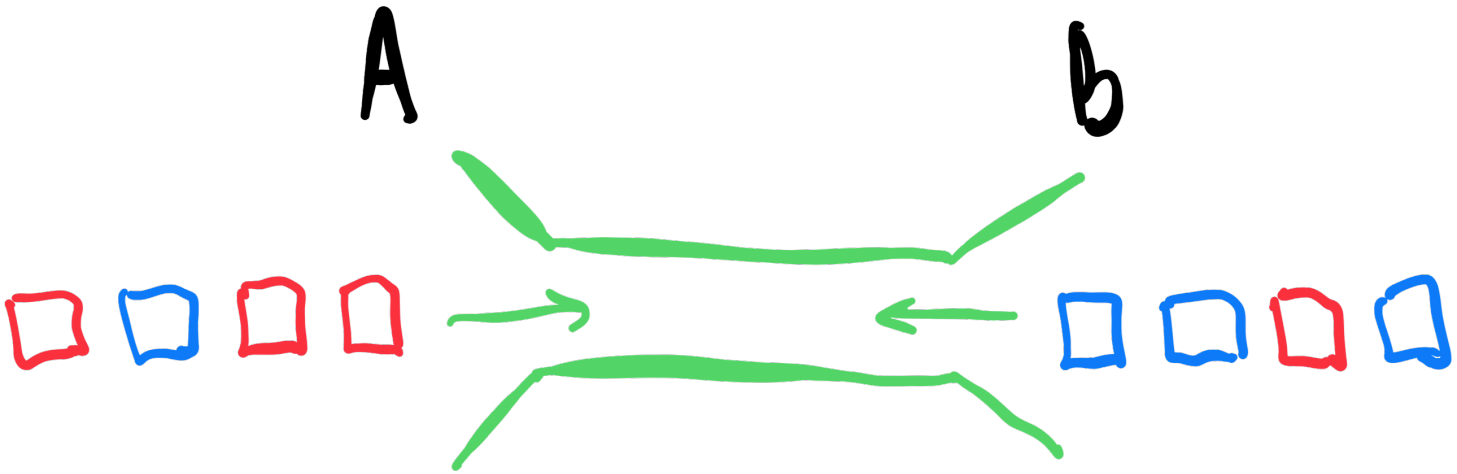
```
import threading

max_connections = 3

semaphore = threading.Semaphore(max_connections) # Инициализируем семафор с максимальным количеством соединений

def database_access():
    semaphore.acquire()
    print("Подключение к базе данных")
    # Выполнение операций с базой данных
    semaphore.release()
    print("Отключение от базы данных")

threads = []
for _ in range(5):
    t = threading.Thread(target=database_access)
    threads.append(t)
```

Задача узкого моста



ТРИ ПРАВИЛА ПРОЕЗДА МАШИН ПО МОСТУ

1) Движение возможно только в одном направлении
2) На мосту могут находиться максимум две машины
3) Машины «скорой помощи» имеют приоритет



Разбор исходных файлов

Файл narrow-bridge-test.c (src/tests/threads) моделирует задачу управления узким мостом для различного числа автомобилей и различных направлений их движения

```
// Test entry point
void test_narrow_bridge(unsigned int num_vehicles_left, unsigned int num_vehicles_right,
                        unsigned int num_emergency_left, unsigned int num_emergency_right)
{
    threads_count_on_start = threads_count();

    narrow_bridge_init();

    create_vehicles(num_vehicles_left, thread_normal_left);
    create_vehicles(num_vehicles_right, thread_normal_right);
    create_vehicles(num_emergency_left, thread_emergency_left);
    create_vehicles(num_emergency_right, thread_emergency_right);

    wait_threads();
}
```

По умолчанию при вызове процедуры test_narrow_bridge() создаются процессы для указанного количества автомобилей и проводится ожидание завершения всех созданных процессов

Моделирование проезда машины по мосту

```
void thread_normal_left(void* arg UNUSED)
{
    one_vehicle(car_normal, dir_left);
}
```

```
void one_vehicle(enum car_priority prio, enum car_direction dir)
{
    arrive_bridge(prio, dir);
    cross_bridge(prio, dir);
    exit_bridge(prio, dir);
}
```

```
void cross_bridge(enum car_priority prio, enum car_direction dir)
{
    msg("Vehicle: %4s, prio: %s, direct: %s, ticks=%4llu",
            thread_current()->name,
            prio == car_emergency ? "emer" : "norm",
            dir == dir_left ? "l -> r" : "l <- r",
            (unsigned long long) timer_ticks ());
    timer_sleep(10);
}
```

В операционной системе Pintos будем представлять один автомобиль одним процессом, который, выполняет следующую процедуру

Ожидание в конце функции моделирует проезд машины по мосту

Моделирование проезда машины по мосту

Файл narrow-bridge.c (src/tests/threads) содержит файлы-заглушки для моделирования проезда машины по мосту

```
// Called before test. Can initialize some synchronization objects.
void narrow_bridge_init(void)
{
    // Not implemented
}

void arrive_bridge(enum car_priority prio UNUSED, enum car_direction dir UNUSED)
{
    // Not implemented. Remove UNUSED keyword if need.
}

void exit_bridge(enum car_priority prio UNUSED, enum car_direction dir UNUSED)
{
    // Not implemented. Remove UNUSED keyword if need.
}
```

Задача

Продумайте систему семафоров, которая будет соответствовать установленным правилам и регулировать движение по мосту без образования заторов (голодания процессов) и разрушения моста

В процедуре arrive_bridge должна оцениваться возможность пересечения автомобилем моста: текущая дорожная ситуация должна быть проанализирована и въезд на мост должен быть заблокирован до тех пор, пока автомобиль не сможет безопасно пересечь мост
Процедура exit_bridge должна соответственно, снимать установленные ограничения на въезд

Ограничения:

1) Нельзя использовать внешнего наблюдатели
2) Нельзя менять аргументы функции
3) Нельзя подсчитывать общее количество машин и после этого принимать решение о проезде
4) Нельзя использовать функции thread_sleep, thread_create

… **И еще много ограничений из методического пособия**

Перед выполнением рекомендуется поставить Pintos без внесенных изменений

Примеры реализаций

```
Executing 'narrow-bridge 4 3 4 3':
(narrow-bridge) begin
(narrow-bridge) Vehicle:    1, prio: norm, direct: l -> r, ticks=  27
(narrow-bridge) Vehicle:    2, prio: norm, direct: l -> r, ticks=  27
(narrow-bridge) Vehicle:   12, prio: emer, direct: l <- r, ticks=  37
(narrow-bridge) Vehicle:   13, prio: emer, direct: l <- r, ticks=  37
(narrow-bridge) Vehicle:   14, prio: emer, direct: l <- r, ticks=  47
(narrow-bridge) Vehicle:    5, prio: norm, direct: l <- r, ticks=  47
(narrow-bridge) Vehicle:    8, prio: emer, direct: l -> r, ticks=  57
(narrow-bridge) Vehicle:    9, prio: emer, direct: l -> r, ticks=  57
(narrow-bridge) Vehicle:   10, prio: emer, direct: l -> r, ticks=  67
(narrow-bridge) Vehicle:   11, prio: emer, direct: l -> r, ticks=  67
(narrow-bridge) Vehicle:    3, prio: norm, direct: l -> r, ticks=  77
(narrow-bridge) Vehicle:    4, prio: norm, direct: l -> r, ticks=  77
(narrow-bridge) Vehicle:    6, prio: norm, direct: l <- r, ticks=  87
(narrow-bridge) Vehicle:    7, prio: norm, direct: l <- r, ticks=  87
(narrow-bridge) end
Execution of 'narrow-bridge 4 3 4 3' complete.
```

Пример корректной работы алгоритма
Машины 1 и 2 подъехали к мосту раньше машин скорой помощи

```
Executing 'narrow-bridge 4 3 4 3':
(narrow-bridge) begin
(narrow-bridge) Vehicle:    1, prio: norm, direct: l -> r, ticks=  27
(narrow-bridge) Vehicle:    2, prio: norm, direct: l -> r, ticks=  27
(narrow-bridge) Vehicle:    8, prio: emer, direct: l -> r, ticks=  37
(narrow-bridge) Vehicle:    9, prio: emer, direct: l -> r, ticks=  37
(narrow-bridge) Vehicle:   10, prio: emer, direct: l -> r, ticks=  47
(narrow-bridge) Vehicle:   11, prio: emer, direct: l -> r, ticks=  47
(narrow-bridge) Vehicle:   12, prio: emer, direct: l <- r, ticks=  57
(narrow-bridge) Vehicle:   13, prio: emer, direct: l <- r, ticks=  67
(narrow-bridge) Vehicle:   14, prio: emer, direct: l <- r, ticks=  77
(narrow-bridge) Vehicle:    3, prio: norm, direct: l -> r, ticks=  87
(narrow-bridge) Vehicle:    4, prio: norm, direct: l -> r, ticks=  97
(narrow-bridge) Vehicle:    5, prio: norm, direct: l <- r, ticks= 107
(narrow-bridge) Vehicle:    6, prio: norm, direct: l <- r, ticks= 117
(narrow-bridge) Vehicle:    7, prio: norm, direct: l <- r, ticks= 127
(narrow-bridge) end
Execution of 'narrow-bridge 4 3 4 3' complete.
```

Пример некорректной работы алгоритма
Например, машины 12 и 13 могут ехать по мосту совместно, а не по очереди