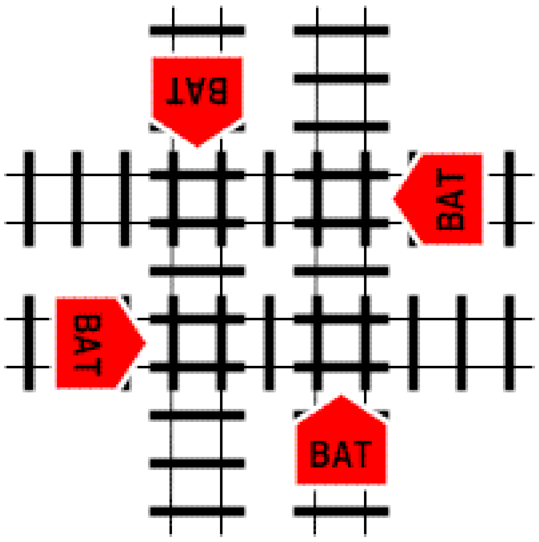
The BATMAN

In a heroic effort to meet increasingly tighter shipping deadlines, a company has employed a Bidirectional Autonomous Trolley (BAT) system to move products from its warehouses to the delivery trucks. Each BAT is a mobile platform that travels on separate tracks back and forth between a warehouse and a truck. Because the goods are fragile, the tracks are perfectly leveled, which requires the placement of level crossings between warehouses. At most one BAT can cross at a time. Traffic at the crossing arriving from the right has the right of way. But this presents a problem, as the company soon found out when a simultaneous shipment of plastic penguins and black umbrellas caused the system to come to a grinding halt. Two BATs with the shipments and two other BATs returning to the warehouses were deadlocked at a level crossing:



In our system each BAT is controlled by a separate thread. It is your task to create BATMAN: a BAT Manager that prevents deadlock at a crossing. Part of the solution is to use one mutex lock for the crossing, so that at most one BAT at a time can pass. The mutex lock will act as a monitor for the BAT operations at the crossing. Besides the mutex lock, we will also need a set of condition variables to control the synchronization of the BATs.

We need a condition variable per BAT to queue BATs arriving from one direction (NorthQueue, EastQueue, SouthQueue, WestQueue). For example, when a BAT from North is already at the crossing, a second BAT from North will have to wait.
Another type of condition variable is needed to let BATs from the right have precedence to cross (NorthFirst, EastFirst, SouthFirst, WestFirst). However, using this rule can cause starvation. To prevent starvation, when a BAT is waiting to cross but BATs continuously arriving from the right have the right of way, we will let a BAT that just passed the crossing signal a waiting BAT on his left.

When deadlock occurs the BAT Manager must signal one of the BATs to proceed, e.g. the BAT from North. You will need a counter for each direction to keep track of the number of BATs waiting in line.

The program must take a string of 'n', 's', 'e', 'w' from the command line indicating a sequence of arrivals of BATs from the four directions. For example:

```
$ ./batman nsewwewn
BAT 4 from West arrives at crossing
BAT 2 from South arrives at crossing
BAT 1 from North arrives at crossing
BAT 3 from East arrives at crossing
DEADLOCK: BAT jam detected, signalling North to go
BAT 1 from North leaving crossing
BAT 3 from East leaving crossing
BAT 2 from South leaving crossing
BAT 4 from West leaving crossing
BAT 6 from East arrives at crossing
BAT 5 from West arrives at crossing
BAT 8 from North arrives at crossing
BAT 5 from West leaving crossing
BAT 6 from East leaving crossing
BAT 8 from North leaving crossing
BAT 7 from West arrives at crossing
BAT 7 from West leaving crossing
```

Note: the ordering of the above arrivals and departures may vary between runs and implementations. You don't need to produce exactly the same output.

To summarize:

- BATs arriving from the same direction line up behind the first BAT already at the crossing;
- BATs arriving from the right always have the right of way (unless the waiting BAT receives a signal to go);
- Deadlock has to be prevented
- Starvation has to be prevented

To implement the solution, analyze the actions of a BAT arriving from a particular direction under different circumstances, i.e. another BAT is already at the crossing, there is a BAT at the right, and/or there is a BAT at the left. Determine what to do when it arrives at the crossing and it must wait in line or wait for the BAT at his right, what to do when it passed the crossing, and what to do when it leaves the crossing.
Consult the manual pages on a Linux machine to understand the following library functions on mutex locks (`pthread_mutex_t`) and condition variables (`pthread_cond_t`):

- *`int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);`*
- *`int pthread_mutex_lock(pthread_mutex_t *mutex);`*
- *`int pthread_mutex_unlock(pthread_mutex_t *mutex);`*
- *`int pthread_mutex_destroy(pthread_mutex_t *mutex);`*

- *int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)*
- *int pthread_cond_signal(pthread_cond_t *cond);*
- *int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)*
- *int pthread_cond_destroy(pthread_cond_t *cond);*

Note that we will use a mutex lock to implement monitor-like functionality, i.e. the BAT operations are part of the monitor.

Addendum

The BATMAN monitor has the following high-level structure:

```
monitor BATMAN
{
  ... // shared data, e.g. counters

  condition variable NorthQueue, EastQueue, SouthQueue, WestQueue,
                     NorthFirst, EastFirst, SouthFirst, WestFirst;

  arrive(BAT b)
  {
    printf("BAT %d from %s arrives at crossing\n", b.num, b.dir);
    ... // code to check traffic in line, use counters, condition variables etc.
  }

  cross(BAT b)
  {
    ... // code to check traffic from the right, use counters, condition variables etc
    sleep(1); // it takes one second for a BAT to cross
  }

  leave(BAT b)
  {
    printf("BAT %d from %s leaving crossing\n", b.num, b.dir);
    ... // code to check traffic, use counters, condition variables etc.
  }

  check()
  {
    ... // the manager checks for deadlock and resolves it
  }
};
```

A BATs execute the following operations:

```
arrive(b);
cross(b);
leave(b);
```

It takes a BAT one second to cross. Use `sleep(1)` to simulate the BAT's progress.

The BAT manager executes the following operations:

```
while BATs are active do
  check();
```

Note that you need a mutex lock to implement the monitor.