Dining Philosophers Semaphore Solution

You can obtain the source to this program with the command:

give kar 440-sem-examples

#define N_PHILOSOPHERS 5

class DiningPhilosophers {
public: // Constructor and destructor
    DiningPhilosophers();
    ~DiningPhilosophers();

public: // Member functions
    void pick_up_forks( int who ); // Called by a thread to pick up both forks.
    void put_down_forks( int who ); // Called by a thread to put down both forks.

private: // Data members
    bool fork[ N_PHILOSOPHERS ]; // Status of each fork
    bool waiting[ N_PHILOSOPHERS ]; // Whether each philosopher is waiting
    bool not_yet_done; // Indication of unfinished work
    Semaphore not_done; // Process with unfinished work
    Semaphore *forks_released[ N_PHILOSOPHERS ]; // Blocks processes waiting for forks
    Semaphore mutex; // Mutual exclusion
}

DiningPhilosophers::DiningPhilosophers(): mutex( 1 ), not_yet_done( false ),
    not_done( 0 ){ // Initially, all forks are available and no one is waiting.
    int i;
    for( i = 0; i < N_PHILOSOPHERS; i += 1 ){
        fork[ i ] = true;
        waiting[ i ] = false;
    }
    for( i = 0; i < N_PHILOSOPHERS; i += 1 ){
        forks_released[ i ] = new Semaphore( 0 );
        assert( forks_released[ i ] != NULL );
    }
}

DiningPhilosophers::~DiningPhilosophers(){ // Must delete the dynamically allocated semaphores
    for( int i = 0; i < N_PHILOSOPHERS; i += 1 ){
        delete forks_released[ i ];
    }
}

void DiningPhilosophers::pick_up_forks( int who ){ // who is the philosopher & left fork, other is the right fork.
    int other( ( who + 1 ) % N_PHILOSOPHERS );
    mutex.P(); // Enter the mutual exclusion
Both forks not available?

Wait!

We're about to block ourselves: if there is someone with unfinished work, unblock them; else open the mutex.

```
while( !( fork[ who ] && fork[ other ] ) ) {
    waiting[ who ] = true;
    if( not_yet_done ){
        not_done.V();
    }
    else {
        mutex.V();
    }
    forks_released[ who ]->P();
    waiting[ who ] = false;
}
```

Now we wait.

```
fork[ who ] = false;
fork[ other ] = false;
if( not_yet_done ){
    not_done.V();
}
else {
    mutex.V();
}
```

Unblocked--check for both of our forks again.

```
forks_released[ who ]->P();
waiting[ who ] = true;
mutex.P();
```

We will pick up our forks, mark them as unavailable.

```
mutex.P();
```

Enter the mutual exclusion

```
who is the philosopher (and the left fork), and other is the right fork and the right neighbor, and
left_neighbor is the left neighbor.
int other( ( who + 1 ) % N_PHILOSOPHERS );
int left_neighbor( ( who + N_PHILOSOPHERS - 1 ) % N_PHILOSOPHERS );
```

Mark the forks as available once again.

```
fork[ who ] = true;
fork[ other ] = true;
if( waiting[ other ] ){
    not_yet_done = true;
    forks_released[ other ]->V();
    not_done.P();
    not_yet_done = false;
}
```

Anyone waiting for the fork on the right?

```
mutex.P();
```

We have to unblock this thread and pass the baton to it, but he's going to pass the baton back to us when he blocks himself again or leaves.

```
if( waiting[ left_neighbor ] ){
    forks_released[ left_neighbor ]->V();
}
else {
    mutex.V();
}
```

Anyone waiting for the fork on the left?

```
mutex.V();
```

We have to unblock this philosopher; we pass the baton to him by just returning without opening the mutex.

```
mutex.V();
```

Otherwise, open the mutex.