PROGRAMMING CONTEST #2 Francesco Taurone

Dining Philosophers

General Idea: The concept behind the proposed algorithm is to nest 2 tables:

1. External: #N_THREADS Seats, resource to be managed → internal table seats.
   - Each philosopher has its own seat, when occupied the phil is "walking/thinking";
   - Differences with respect to the Classic Dining Phil table:
     - Each phil can reach any internal seat
     - Each phil does not have to wait for his left and right neighbours, rather just for a seat to be free (n_sit<N_SEATS)
2. Internal: #N_SEATS Seats, resource to be managed → chopsticks.
   - Classic Dining Phil problem, but in this case the seat might be also empty, waiting to be occupied.

The strategy was to solve the 2 tables as if they were 2 different problems, then to merge the 2 by adjusting the references.

MONITOR

- pthread_mutex_t m;
- pthread_cond_t phil_cv[N_THREADS];

  Instead of the 1 queue strategy, here there is 1 queue per phil. This decision is based on the strategy implemented for avoiding starvation, that in this case makes use of differentiated queues.
- int where_sit[N_THREADS];
- int who_sit[N_SEATS];

  One of the 2 could have been avoided. But the implementation of both avoids the intensive use of small cycles, which improves clearness.
- int who_was_polite[N_THREADS];

  Needed for implementing starvation handling in this algorithm
- int n_wait_chop;

  Number of phil waiting for chopsticks. Used in inner table
- state_t state_phil[N_THREADS];

  For keeping track of the behaviour of each phil.
- long int eating_situation[N_THREADS];

  For monitoring the eating situation while debugging
- int n_sit;

  Number of phil seated, for checking if there are vacant seats

STATES

typedef enum {THINKING, HUNGRY_STAND, HUNGRY_SIT, EATING, POLITE} state_t;

Here each Phil can be in these states (sequentially). HUNGRY_STAND is when the phil is hungry but not seated, therefore waiting for a seat, checking for it or object to some "polite" Phil (refer to Starvation section). HUNGRY_SIT is when the phil is hungry while seated, waiting for the chopsticks.
POLITE state is used for starvation handling.
A phil is FINISHED when he finished the thinking-eating cycles.

DEADLOCK AND CONCURRENCY

By means of the monitor implementation, Deadlock conditions are prevented by making each phil waiting for both chopsticks to be free. Once "eating" phase is finished, each phil signals the neighbours. Concurrency is possible, since up to 2 seated phil can eat at the same time uder these conditions.

STARVATION

For preventing starvation from happening, the concept of POLITE state is introduced in both tables.

- Internal table Starvation

Let’s suppose an inner table of such kind: (seats)CAB(seats), A eating, at T=1 finishes
Starvation was noticed to happen mainly due to the fact that if A signals the neighbours at distance 1, it might happen that only one of the 2, maybe B, can start eating while the other one C needs to wait again. Then, when B has finished, he signals the neighbours at distance 1, in which C is not included.
Therefore, even if B and C were signalled at the same time T=1, it might happen that the neighbours at distance 1 of B signalled at time T>1 have higher priority than C. Might be said that it resembles the priority inversion problem in some ways. Then the solution might be to increase the Neighbourhood size to 2, signaling first the most distant ones.

However, given that all the signals are in the same function, just signaling more doesn’t guarantee that C will be the first among the neighbours to be signaled. So, B must check C, and if C is still HUNGRY_SIT, B must wait al let C check if his chopsticks are available. If yes, C should eat than signal back B, who was polite with him (from here the syntax who_was_polite[]), if no just signal back. Then B, woken up by C, can proceed signaling hungry neighbours from the most distant once, with ne remark that the neighbours at distance 1 are treated classically, so that no phil can be polite with the ones with whom he is sharing chopsticks.

This approach can be extended to any feasible size NEIGH_INT, defined for the preprocessor

- External Table Starvation

Similarly, each phil is polite with his neighbours (also the ones at distant 1 are included). This time once the seated phil wants to stand up, before letting anyone take his seat he check if his neighbours want that. In that case, he neither signals anyone else, nor decreases the n_seated, letting only the HUNGRY_STAND neighbour substitute him.

In this way, all the phil belong to different neighbourhood, meaning that there is always going to be a neighbourhood in which is philosopher is prioritized. This has proven to limit the starvation issue.

A proposed more robust robust alternative might me to introduce an additional state EMERGENCY and a table with an entry for each phil. At each phil’s operation, the corresponding value might be increased and after a certain threshold that philosopher gets a privileged path across the table for eating before any other. The entry should be reset to zero every time the phil eats. The seated phil should pick_up only with the additional condition that no phil in emergency state is beside them.

This solution was implemented,but there was no time to debug it properly, as well as the introduction of a FINISHED state to ensure that the neighbours who have finished their cycle are skipped in counting the distance.