1.DINING PHILOSOPHER PROBLEM

The dining philosophers problem is summarized as five philosophers sitting at a table doing one of two things: eating or thinking. While eating, they are not thinking, and while thinking, they are not eating. The five philosophers sit at a circular table with a large bowl of spaghetti in the center. A fork is placed in between each pair of adjacent philosophers, and as such, each philosopher has one fork to his left and one fork to his right. As spaghetti is difficult to serve and eat with a single fork, it is assumed that a philosopher must eat with two forks. Each philosopher can only use the forks on his immediate left and immediate right.

The dining philosophers problem is sometimes explained using rice and chopsticks rather than spaghetti and forks, as it is more intuitively obvious that two chopsticks are required to begin eating.
The philosophers never speak to each other, which creates a dangerous possibility of deadlock when every philosopher holds a left fork and waits perpetually for a right fork (or vice versa).

Originally used as a means of illustrating the problem of deadlock, this system reaches deadlock when there is a 'cycle of unwarranted requests'. In this case philosopher P1 waits for the fork grabbed by philosopher P2 who is waiting for the fork of philosopher P3 and so forth, making a circular chain.

Starvation (and the pun was intended in the original problem description) might also occur independently of deadlock if a philosopher is unable to acquire both forks because of a timing problem. For example there might be a rule that the philosophers put down a fork after waiting five minutes for the other fork to become available and wait a further five minutes before making their next attempt. This scheme eliminates the possibility of deadlock (the system can always advance to a different state) but still suffers from the problem of livelock. If all five philosophers appear in the dining room at exactly the same time and each picks up their left fork at the same time the philosophers will wait five minutes until they all put their forks down and then wait a further five minutes before they all pick them up again.

In general the dining philosophers problem is a generic and abstract problem used for explaining various issues which arise in problems which hold mutual exclusion as a core idea. The various kinds of failures these philosophers may experience are analogous to the difficulties that arise in real computer programming when multiple programs need exclusive access to shared resources. These issues are studied in the branch of Concurrent Programming. The original problems of Dijkstra were related to external devices like tape drives. However, the difficulties studied in the Dining Philosophers problem arise far more often when multiple processes access sets of data that are being updated. Systems that must deal with a large number of parallel processes, such as operating system kernels, use thousands of locks and synchronizations that require strict adherence to methods and protocols if such problems as deadlock, starvation, or data corruption are to be avoided.

WAITER SOLUTION:

A relatively simple solution is achieved by introducing a waiter at the table. Philosophers must ask his permission before taking up any forks. Because the waiter is aware of which forks are in use, he is able to arbitrate and prevent deadlock. When four of the forks are in use, the next philosopher to request one has to wait for the waiter's permission, which is not given until a fork has been released. The logic is kept simple by specifying that philosophers always seek to pick up their left hand fork before their right hand fork (or vice versa).

To illustrate how this works, consider the philosophers are labelled clockwise from A to E. If A and C are eating, four forks are in use. B sits between A and C so has neither fork available, whereas D and E have one unused fork between them. Suppose D wants to eat. Were he to take up the fifth fork, deadlock becomes likely. If instead he asks the waiter and is told to wait, we can be sure that next time two forks are released there will certainly be at least one philosopher who could successfully request a pair of forks. Therefore deadlock cannot happen.

RESOURCE HIERARCHY SOLUTION:

Another simple solution is achieved by assigning a partial order to the resources (the forks, in this case), and establishing the convention that all resources will be requested in order, and released in reverse order, and that no two resources unrelated by order will ever be used by a single unit of work at the same time. Here, the resources (forks) will be numbered 1 through 5, in some order, and each unit of work (philosopher) will always pick up the lower-numbered fork first, and then the higher-numbered fork, from among the two forks he plans to use. Then, he will always put down the higher numbered fork first, followed by the lower numbered fork. In this case, if four of the five philosophers simultaneously pick up their lower-numbered fork, only the highest numbered fork will remain on the table, so the fifth philosopher will not be able to pick up any fork. Moreover, only one philosopher will have access to that highest-numbered fork, so he will be able to eat using two forks. When he finishes using the forks, he will put down the highest-numbered fork first, followed by the lower-numbered fork, freeing another philosopher to grab the latter and begin eating.
While the resource hierarchy solution avoids deadlocks, it is not always practical, especially when the list of required resources is not completely known in advance. For example, if a unit of work holds resources 3 and 5 and then determines it needs resource 2, it must release 5, then 3 before acquiring 2, and then it must re-acquire 3 and 5 in that order. Computer programs that access large numbers of database records would not run efficiently if they were required to release all higher-numbered records before accessing a new record, making the method impractical for that purpose.
This is often the most practical solution for real world Computer Science problems; by assigning a constant hierarchy of locks, and by enforcing the ordering of obtaining the locks this problem can be avoided.

MONITOR SOLUTION:

The example below shows a solution where the forks are not represented explicitly. Philosophers can eat if none of their neighbors are eating. This is comparable to a system where philosophers that cannot get the second fork must put down the first fork before they try again.
In the absence of locks associated with the forks, philosophers must ensure that the decision to begin eating is not based on stale information about the state of the neighbors. If philosopher B sees that A is not eating, then turns and looks at C, A could begin eating while B looks at C. This solution avoids this using a single mutual exclusion lock, not associated with the forks, but with the decision procedures that can change the states of the philosophers. This is ensured by the monitor. The procedures test, pickup and putdown are local to the monitor and share a mutual exclusion lock. Notice that processes calling WAITC(x) release the lock and wait on the variable x until another process calls SIGNALC(x) on the same variable. When the process that called WAITC resumes, it will have reacquired the lock.
Notice that this example does not tackle the starvation problem. For example, philosopher 2 can wait forever on condition variable can_eat[2] if the eating periods of philosophers 1 and 3 always overlap.
To also guarantee that no philosopher starves, one could keep track of the number of times a hungry philosopher cannot eat when his neighbors put down their forks. If this number exceeds some limit, the state of the philosopher could change to Starving, and the decision procedure to pick up forks could be augmented to require that none of the neighbors are starving.
A philosopher that cannot pick up forks because a neighbor is starving, is effectively waiting for the neighbor's neighbor to finish eating. This additional dependency reduces concurrency. Raising the threshold for transition to the Starving state reduces this effect.

EXAMPLE SOLUTION:

```
PROGRAM d_p;
   CONST
      DoomsDay = FALSE;
   MONITOR dining_philosophers;  // Start of monitor declaration
      CONST
         Eating   = 0;
         Hungry   = 1;
         Thinking = 2;
      VAR
         i       : INTEGER; // init loop variable
         state   : ARRAY [0..4] OF INTEGER; // Eating, Hungry, Thinking
         can_eat : ARRAY [0..4] OF CONDITION; // one for each Philosopher
         // place for Hungry Ph to wait until chopsticks become available

      PROCEDURE test(k : INTEGER);
      BEGIN
         IF (state[k] = Hungry)
            AND (state[(k+4) MOD 5] <> Eating)
            AND (state[(k+1) MOD 5] <> Eating) THEN
         BEGIN
            state[k] := eating;
            SIGNALC(can_eat[k]); // End the wait if any
         END;
      END;

      PROCEDURE pickup(i: INTEGER);
      BEGIN
         state[i] := Hungry;
         WRITELN('philosopher ',i,' hungry');
         test(i); // are my neighbors eating?
         IF state[i] <> Eating THEN
            WAITC(can_eat[i]);  // wait until they finish eating
         WRITELN('philosopher ',i,' eating');
      END;

      PROCEDURE putdown(i : INTEGER);
      BEGIN
         state[i] := Thinking;
         WRITELN('philosopher ',i,' thinking');
         test( (i+4) MOD 5); // give left neighbor chance to eat
         test( (i+1) MOD 5); // give right neighbor chance to eat
      END;

   BEGIN // initialize monitor
      FOR i := 0 TO 4 DO state[i] := Thinking;
   END;  // end of monitor definition

   PROCEDURE philosopher(i : INTEGER);
   BEGIN
      REPEAT
         pickup(i);  // pick up chopsticks
         putdown(i); // put down chopsticks
      UNTIL DoomsDay;
   END;

BEGIN // main program
   COBEGIN  // start all five processes at once
      philosopher(0); philosopher(1); philosopher(2);
      philosopher(3); philosopher(4);
   COEND;
END
```