Q1- Cigarette smokers problem

The problem involves synchronization of four tasks: three smokers and one agent. The smokers repeatedly perform actions of waiting for ingredients, making a cigarette and then smoking it. To make and smoke a cigarette three ingredients are needed: tobacco, paper, and matches. The agent has an infinite supply of all the three ingredients; however, each of the smokers has an infinite supply of only one of the ingredients. One smoker has tobacco, another has paper, and the third has matches. The agent repeatedly chooses two ingredients at random and makes them available to the smokers. A smoker that has the complementary ingredient picks up the two ingredients along with signaling it to the agent, makes a cigarette, and smokes it. For example, if the agent made available tobacco and paper, then the smoker with matches picks up both ingredients. Picking up the ingredients is signaled to the agent so that it can choose and make available the next pair of ingredients. In the problem under consideration the agent represents for example a computer operating system allocating resources to tasks that need them. Assume the following declarations of binary semaphores:

```
agentS: semaphore := (1, ∅);
tobacco, paper, matches: semaphore := (0, ∅);
```

Q2- In a busy kitchen, there are three workers, each responsible for preparing a specific part of a hamburger.
Vegetable Worker: Prepares the vegetables,
Beef Worker: Cooks the beef patty.
Bread Worker: Prepares the bread.
Each worker operates independently and starts working at the same time. The Burger assembler can only assemble the hamburger once all three parts (vegetables, beef patty, and bread) are ready. Write a multithreaded program to model this process. Ensure that the tasks are completed in the correct order based on the dependencies listed above.

Q3- Given a competition between 2 generator agents, named ‘Agent1’ and ‘Agent2’. Each generator agent generates a random number between 1 and 10, and saves this number in a common variable, called ‘Num’. This variable is shared between all agents. Another thread, named ‘collector’, checks the value in ‘Num’; if this number is even, the collector increases the even counter by 1; otherwise it increases the odd counter. This process continues until one of the counters reaches 100.
Note that the value in ‘Num’ should be updated by only one agent, either ‘Agent1’ or ‘Agent2’, at a time, i.e. ‘Agent1’ and ‘Agent2’ cannot update ‘Num’ at the same time. In addition, the value ‘Num’ should not be updated until the collector increases its even or odd counter.
Write **a parallel program** (using multi-threading) to simulate the 2 agents and the collector. The program should also display who reaches one 100, the even counter or the odd counter. To solve this task, you need to identify the critical sections in the program and implement them using techniques you learned in the lab.

Q4-Dining philosophers problem

The dining philosophers problem is a classic problem of synchronization, in which tasks require simultaneous access to more than one resource. This problem is formulated as follows. Five philosophers are sitting at a round table spending time on doing two things: thinking and eating. At the center of the table there is a bowl of spaghetti constantly replenished, and five plates one in front of each of the philosophers. To eat, a philosopher needs two forks lying on the table to his/her right and left. A philosopher can pick up either the right and then the left fork (or vice versa)—if they are available, that is if they have not been picked up by neighbor philosophers—but not both at a time. There are only five forks and each of them can only be used at one time by one philosopher. So at most two philosophers not sitting at the table next to each other can eat simultaneously

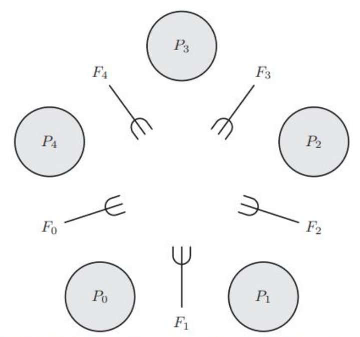
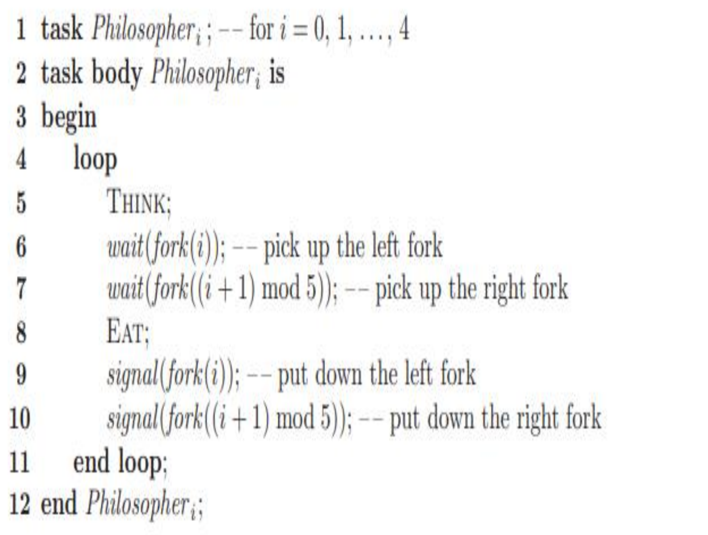


Figure 1.9. The dining philosophers, $P_0 \dots P_4$ – philosophers, $F_0 \dots F_4$ – forks.

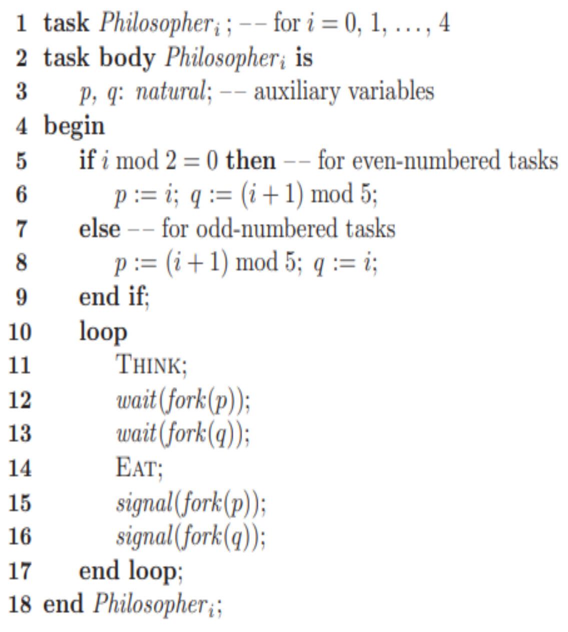
Deadlock by not allowing philosophers to release a fork before picking up the other. Philosophers can pick up a fork but must wait until they have both forks to eat. This can lead to a situation where all philosophers have one fork, and they are all waiting for the second one, causing a deadlock.



```
task $Philosopher_i$; -- for $i = 0, 1, \dots, 4$
task body $Philosopher_i$ is
begin
    loop
        THINK;
        $wait(fork(i))$; -- pick up the left fork
        $wait(fork((i+1) \bmod 5))$; -- pick up the right fork
        EAT;
        $signal(fork(i))$; -- put down the left fork
        $signal(fork((i+1) \bmod 5))$; -- put down the right fork
    end loop;
end $Philosopher_i$;
```

Figure 1.10. Solving the problem of the dining philosophers where deadlock may occur.

This solution demonstrates an asymmetric approach where philosophers have different rules for acquiring forks based on their index. It ensures that philosophers don't deadlock and can follow the asymmetric rule for acquiring forks to eat.



```
task $Philosopher_i$; -- for $i = 0, 1, \dots, 4$
task body $Philosopher_i$ is
    $p$, $q$: natural; -- auxiliary variables
begin
    if $i \bmod 2 = 0$ then -- for even-numbered tasks
        $p := i$; $q := (i+1) \bmod 5$;
    else -- for odd-numbered tasks
        $p := (i+1) \bmod 5$; $q := i$;
    end if;
    loop
        THINK;
        $wait(fork(p))$;
        $wait(fork(q))$;
        EAT;
        $signal(fork(p))$;
        $signal(fork(q))$;
    end loop;
end $Philosopher_i$;
```

Figure 1.12. An asymmetric solution to the dining philosophers problem.