The operating systems literature is full of interesting problems that have been widely discussed and analyzed using a variety of synchronization methods. In the following sections we will examine four of the better-known problems.

5.1 The Dining Philosophers Problem

Since 1965 (Dijkstra posed and solved this synchronization problem), everyone inventing a new synchronization primitive has tried to demonstrate its abilities by solving the dining philosophers problem. The problem can be stated quite simply as follows. Five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates, there is only one fork.

The life of a philosopher consists of alternate periods of eating and thinking. This is something of an abstraction, even for philosophers, but the other activities are irrelevant here. When a philosopher gets hungry, she tries to acquire her left and right forks, one at a time, in either order. If successful in acquiring two forks, she eats for a while, then puts down the forks, and continues to think.

It has been pointed out that the two-fork requirement is somewhat artificial; perhaps we should switch from Italian food to Chinese food, replacing rice for spaghetti and chopsticks for forks.

The key question is: Can you write a program for each philosopher that does what it is supposed to do and never gets stuck? There is a obvious solution.
5.1 The Dining Philosophers Problem

```c
#define N 5;
void philosopher(int i) {
    while(TRUE) {
        think();
        take_fork(i);
        take_fork((i+1)%5);
        eat();
        put_fork(i);
        put_fork( (i+1)%5);
    }
}
```

The procedure `take_fork` waits until the specified fork is available and then seizes it. Unfortunately, the obvious solution is wrong. Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a deadlock.

What about semaphores?

5.1 The Dining Philosophers Problem (poor solution)

```c
shared binary semaphore fork[5] = 1;
void philosopher(int i) {
    while(TRUE) {
        think();
        down(fork[i]);
        down(fork[(i+1)%5]);
        eat();
        up(fork[i]);
        up(fork[(i+1)%5]);
    }
}
```

The same problem is present. We could modify the program so that after taking the left fork, the program checks to see if the right fork is available. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process. With a little bit of bad luck, all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks, waiting, picking up their left forks again at the same time, and so on, forever.

5.1 The Dining Philosophers Problem

A situation like this (in which all the programs continue to run indefinitely but fail to make any progress) is called starvation. It is called starvation even when the problem does not occur in an Italian or a Chinese restaurant.

Another idea is might be waiting a random time after failing to acquire the right-hand fork. In nearly all applications trying again later is not a problem.

For example, in the popular Ethernet local area network, if two computers send a packet at the same time, they crash on the net, then each one waits a random time and tries again. If we would have no option, this solution would be accepted usable.

However, we have one solution that always works and cannot fail due to an unlikely series of random numbers.

Let us try a mutex semaphore.
Another improvement is to protect the five statements following the call to think by a binary semaphore. Before starting to acquire forks, a philosopher would do a down on mutex. After replacing the forks, she would do an up on mutex.

From a theoretical viewpoint, this solution is sufficient. From a practical one, it has a performance bug: only one philosopher can be eating at any instant.

Using mutex, we avoid starvation but now only one philosopher can be active. No two of them can eat at the same time.

With five forks available, we should be able to allow two philosophers to eat at the same time.

If we use mutex while the philosopher take the forks and add some variables to control her neighbors, it may be better solution. Also when putting the forks, to get active her hungry neighbors would be the best solution.

The program uses an array of semaphores, one per philosopher, so hungry philosophers can block if the needed forks are busy. Note that each process runs the procedure philosopher as its main code, but the other procedures, take_forks, put_forks, and test, are ordinary procedures and not separate processes.
A data object is shared among several concurrent processes. **Readers** Processes that only want to read the shared object. **Writers** Processes that want to update (read and write) the shared object.

Any number of readers may access the data at one time, but writers must have exclusive access.

How do we program the reader and writer?

### 5.2 The Readers and Writers Problem

- **No reader** will be kept waiting unless a writer is already updating the data, (sometimes called the first reader-writer problem).
- **If a writer** is waiting to access the data, no new readers may start reading, (sometimes called the second reader-writer problem).

Both of these approaches may lead to starvation.

```c
shared binary semaphore mutex = 1;
shared binary semaphore db = 1;
int readcount = 0;

void reader () {
    down(mutex);
    readcount++;
    if (readcount == 1) then down(db);
    up(mutex);
    CRITICAL REGION to read
    down(mutex);
    readcount--;
    if (readcount == 0) then up(db);
    up(mutex);
}

void writer () {
    down(db);
    CRITICAL REGION to write
    up(db);
}
```

---

**Diagram:**

- Reader
  - Readers can enter
  - Reader
  - Database
  - Reader

- Writer
  - Writers must wait
  - Access to database

---

**Diagram:**

- Reader
  - Reader
  - Writer
  - Database

---

**Diagram:**

- Reader
  - Reader
  - Writer
  - Database

---

**Diagram:**

- Reader
  - Reader
  - Writer
  - Access to database

---

**Diagram:**

- Reader
  - Reader
  - Writer
  - Access to database
  - Database

---
5.2 The Readers and Writers Problem

In fact, the solution may starve writers in the queue. Therefore, the new solution is sometimes proposed, which adds the constraint that no thread shall be allowed to starve; that is, the operation of obtaining a lock on the shared data will always terminate in a bounded amount of time.

5.3 The Sleeping Barber Problem

Another classical IPC problem takes place in a barber shop. A barbershop consists of a waiting room with $n$ chairs, and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the barber and the customers.

5.3 The Sleeping Barber Problem

Our solution uses three semaphores, customers (counts waiting customers), barbers (the number of barbers who are idle), and mutex (mutual exclusion). We also need a variable, waiting, which also counts the waiting customers. The reason for having waiting is that there is no way to read the current value of a semaphore. In this solution, a customer entering the shop has to count the number of waiting customers. If it is less than the number of chairs, he stays; otherwise, he leaves.
5.3 The Sleeping Barber Problem

When the barber shows up for work in the morning, he executes the procedure barber, causing him to block on the semaphore customers because it is initially 0. The barber then goes to sleep. He stays asleep until the first customer shows up. When a customer arrives, he executes customer, starting by acquiring mutex to enter a critical region. If another customer enters shortly thereafter, the second one will no be able to do anything until the first one has released mutex.

5.4 The Cigarette Smokers

Four threads are involved: an agent and three smokers. To make a cigarette, the ingredients are tobacco, paper, and matches. The smokers loop forever, first waiting for ingredients, then making and smoking cigarettes. We assume that the agent has infinite supply of all three ingredients, and each smoker has infinite supply of one of the ingredients; that is, one smoker has matches, another has paper, and the third has tobacco.

The agent uses the following semaphores:

- binary semaphore agentSem = 1;
- semaphore tobacco = 0;
- semaphore paper = 0;
- semaphore match = 0;

The agent is actually made up of three concurrent threads, Agent A, Agent B and Agent C. Each one does a down on agentSem; each time agentSem is signaled, one of the Agents wakes up and provides ingredients by doing an up on two semaphores.

The customer then checks to see if the number of waiting customers is less than the number of chairs. If not, he releases mutex and leaves without a haircut. If there is an available chair, the customer increments the integer variable, waiting. Then he does an up on the semaphore customers, thus waking up the barber. At this point, the customer and the barber are both awake. When the customer releases mutex, the barber grabs it, does some housekeeping, and begins the haircut.

The agent repeatedly chooses two different ingredients at random and makes them available to the smokers. Depending on which ingredients are chosen, the smoker with the complementary ingredient should pick up both resources and proceed. For example, if the agent puts out tobacco and paper, the smoker with the matches should pick up both ingredients, make a cigarette, and then signal the agent.

void agent_A() {
    down(agentSem);
    up(tobacco);
    up(paper);
}

void Agent_B() {
    down(agentSem);
    up(paper);
    up(match);
}

void Agent_C() {
    down(agentSem);
    up(tobacco);
    up(match);
}

void smoker_matches() {
    down(tobacco);
    down(paper);
    up(agentSem);
}

void smoker_tobacco() {
    down(paper);
    down(match);
    up(agentSem);
}

void smoker_paper() {
    down(match);
    down(tobacco);
    up(agentSem);
}
What’s wrong with this solution? The problem with this solution is the possibility of deadlock.

Imagine that when there was nothing on the table, the agent_A puts out tobacco and paper. Just after smoker_matches used the last tobacco, scheduler changes into smoker_tobacco. Unfortunately it uses the last paper on the table and it leads a DEADLOCK! Four processes must wait forever.

The three smoker processes will make a cigarette and smoke it. If they can’t make a cigarette, then they will go to sleep. The agent process will place two items on the table, and wake up the appropriate smoker, and then go to sleep.

The smoker immediately sleeps. When the agent puts the two items on the table, then the agent will wake up the appropriate smoker. The smoker will then grab the items, and wake the agent.

The variables and semaphores are:

```
semaphore tobacco = 0; //counter
semaphore paper = 0; //counter
semaphore match = 0; //counter
binary semaphore tobaccoSem = 0;
binary semaphore paperSem = 0;
binary semaphore matchSem = 0;
binary semaphore mutex = 1;
```

```c
void agent() {
  while(TRUE) {
    down(mutex);
    randNum = rand(3);
    if (randNum == 1) {
      up(tobacco);
      up(paper);
      up(matchSem);
    } elseif (randNum == 2) {
      up(tobacco);
      up(match);
      up(paperSem);
    } else {
      up(match);
      up(paper);
      up(tobaccoSem);
    }
    …
    up(mutex);
    down(agentSem);
  }
}
```