Classical Synchronization Problems

Introduction

This handout states three classical synchronization problems that are often used to compare language constructs that implement synchronization mechanisms and critical sections.

The Producer-Consumer Problem

In this problem, two processes, one called the *producer* and the other called the *consumer*, run concurrently and share a common buffer. The producer generates items that it must pass to the consumer, who is to consume them. The producer passes items to the consumer through the buffer. However, the producer must be certain that it does not deposit an item into the buffer when the buffer is full, and the consumer must not extract an item from an empty buffer. The two processes also must not access the buffer at the same time, for if the consumer tries to extract an item from the slot into which the producer is depositing an item, the consumer might get only part of the item. Any solution to this problem must ensure none of the above three events occur.

A practical example of this problem is electronic mail. The process you use to send the mail must not insert the letter into a full mailbox (otherwise the recipient will never see it); similarly, the recipient must not read a letter from an empty mailbox (or he might obtain something meaningless but that looks like a letter). Also, the sender (producer) must not deposit a letter in the mailbox at the same time the recipient extracts a letter from the mailbox; otherwise, the state of the mailbox will be uncertain.

Because the buffer has a maximum size, this problem is often called the *bounded buffer problem*. A (less common) variant of it is the unbounded buffer problem, which assumes the buffer is infinite. This eliminates the problem of the producer having to worry about the buffer filling up, but the other two concerns must be dealt with.

The Readers-Writers Problem

In this problem, a number of concurrent processes require access to some object (such as a file.) Some processes extract information from the object and are called *readers*; others change or insert information in the object and are called *writers*. The Bernstein conditions state that many readers may access the object concurrently, but if a writer is accessing the object, no other processes (readers or writers) may access the object. There are two possible policies for doing this:

First Readers-Writers Problem. Readers have priority over writers; that is, unless a writer has permission to access the object, any reader requesting access to the object will get it. Note this may result in a writer waiting indefinitely to access the object.

Second Readers-Writers Problem. Writers have priority over readers; that is, when a writer wishes to access the object, only readers which have already obtained permission to access the object are allowed to complete their access; any readers that request access after the writer has done so must wait until the writer is done. Note this may result in readers waiting indefinitely to access the object.

So there are two concerns: first, enforce the Bernstein conditions among the processes, and secondly, enforcing the appropriate policy of whether the readers or the writers have priority.

A typical example of this occurs with databases, when several processes are accessing data; some will want only to read the data, others to change it. The database must implement some mechanism that solves the readers-writers problem.

The Dining Philosophers Problem

In this problem, five philosophers sit around a circular table eating spaghetti and thinking. In front of each philosopher is a plate and to the left of each plate is a fork (so there are five forks, one to the right and one to the left of each philosopher's plate; see the figure). When a philosopher wishes to eat, he picks up the forks to the right and to the left of his plate. When done, he puts both forks back on the table. The problem is to ensure that no philosopher will be allowed to starve because he cannot ever pick up both forks.

There are two issues here: first, deadlock (where each philosopher picks up one fork so none can get the second) must never occur; and second, no set of philosophers should be able to act to prevent another philosopher from ever eating. A solution must prevent both.

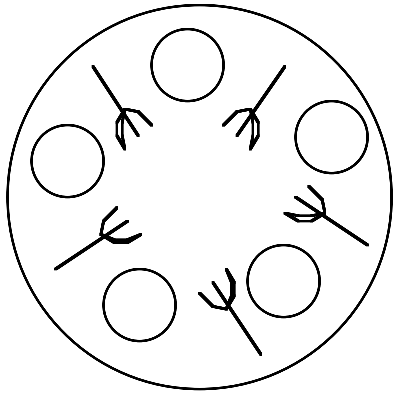

Figure. The Dining Philosopher's Table