Systems Programming and Operating Systems	**Induprakas Kodukula**
CS414, Summer 1996	**Class#9, 7/24/96**

1 Class #9

This section looks at two remaining aspects of the critical section problem. First, a set of standard problems to deal with process synchronization are explored. Next, some higher level constructs than the ones examined so far are examined to provide easier solutions to the synchronization problem at the application level.

1.1 Classic Problems in Synchronization

There are three *classic* problems that are used to validate any solution or approach that attempts to provide a solution to the synchronization problem. These three are dealt with in this section.

1.1.1 Bounded Buffer problem

The bounded buffer problem consists of a buffer with a fixed number of entries. A *producer* process attempts to produce data to insert it into the bounded buffer. A *consumer* process attempts to consume the data from the bounded buffer. There is a problem of mutual exclusion involved in the sense that the producer cannot be writing a data item at the same time that the consumer is reading a data item. In addition, the boundedness of the buffer imposes some additional conditions. For example, the producer cannot insert an element into a full bounded buffer. Similarly, the consumer cannot read a data item from a buffer which is empty. Semaphores can be used to implement a solution to the bounded buffer problem as follows: Three semaphores, *lock*, *filled* and *empty* will be used. *lock* is used to ensure mutual exclusion. *filled* is used to count the number of filled locations and *empty* is used to count the number of empty locations. If the number of filled locations is 0, the consumer must block. If the number of empty locations

```
Procedure producer(item)
  wait(empty)
  wait(lock)
  ..insert item
  signal(lock)
  signal(filled)
end producer

Procedure consumer(item)
  wait(filled)
  wait(lock)
  ..consume item
  signal(lock)
  signal(empty)
end consumer
```

1.1.2 Reader's and Writer's Problem

The reader's and writer's problem can be defined as follows. Given a database, there are multiple processes that either *read* from or *write* to the database. Because readers do not modify the data they read, it is possible for multiple reader processes to be in the database at the same time. However, it is not possible for multiple writer processes to be in the database at the same time. In addition, it is not possible for reader processes to be around in the data base when a writer process is around, since the reader processes may end up with stale data. For a solution to this process, again semaphores will be used in addition to a *readcount* variable. The purpose of the *readcount* variable is to keep track of the number of read processes in the system. Writer process mutual exclusion can be expressed quite simply by using a semaphore called *write*. The purpose of the *readcount* variable is to let a reader process determine if there are any other reader processes in the system. If there are, then the reader can simply enter the system. If the reader process attempts to enter when there are no other reader processes in the system, it must first wait for any writer process currently executing in the system to leave. This is done by waiting on the same semaphore that the writer process uses. Similarly, if a reader process is the last one to leave the database, it has the responsibility of signalling the semaphore that the writer process uses to enable it to enter the database. Finally, the updating of the *readcount* variable is itself a critical section between variour reader processes and must be done atomically using another semaphore called *mutex*. All this can be implemented as follows:

```
Procedure writer()
  wait(write);
  .. write into database
  signal(write);
end writer

Procedure reader()
  wait(mutex);
  readcount = readcount + 1;
  if (readcount == 1)  then
    wait(write);
  endif
  signal(mutex);
  .. read from database
  wait (mutex);
  reacount = readcount - 1;
  if (readcount == 0) then
    signal(write);
  end if
  signal (mutex);
end reader
```

1.1.3 The dining philosopher's problem

The dining philosopher's problem is a very important and classical problem in the context of achieving mutual exclusion and avoiding deadlock at the same time. The problem can be defined very easily. There are some number (say 5) of philosophers sitting at a round table. Between any two philosophers, lies a chopstick. A philosopher does one of two things: thinking or eating. When a philospher thinks, he does nothing. However, to be able to eat, he has two acquire two chopsticks: the one on his left and the one on his right. Once he finishes eating, he can put down the chopsticks and one of his neighbors may start to eat if they so desire. A simple attempt at a solution for this problem is to use an array of semaphores, chopsticks[0..4], and try to use the following discipline for each philosopher:

```
Procedure attempt_to_eat(i)
  wait (chopstick[i]);
  wait (chopstick[i+1 mod 5]);
  ....eat...
  signal(chopstick[i]);
  signal(chopstick[i+1]);
end attempt_to_eat
```

This procedure suffers from the possibility of deadlocks. If all the philosophers want to eat at the same time and all of them pick up the fork on their left, then they will simply keep waiting for ever. There are a number of ways of preventing deadlock. For instance, the number of philosphers at the tabke could be limited to 4, or assymetry could be introduced between even and odd philosophers. Another solution would be for each philosopher to test if the left and the right chopsticks are free (this has to be done in a critical section), and enter the eating procedure only when both the chopsticks are free.

1.2 Higher Level Constructs

Having examined three of the problems that are representative of synchronization problems, and having seen some of the complexity of the solutions involved, next it is worthwhile to see some higher level constructs that are available to deal with the issue of synchronization. Two such commonly available constructs will be studied in this section.

1.2.1 Conditional Critical Section

A conditional critical section mechanism requires programs using this mechanism to employ the following discipline. It requires all shared variables to be declared specially. In addition, it requires that all accesses to the shared variables be inside specially marked region statements. For example, a shared variable called v may be declared as

```
  var v: int shared
```

and a conditional critical section using this variable can be written as

```
  region v when (cond) do stmt
```

The semantics of this region construct is that when one process is inside its region accessing v, it is the only process to be able to access v. The compiler generates the right code to satisfy this condition at all times. Using conditional critical sections and a structure(record) definition for the buffer variable, a bounded buffer problem can now be solved as follows:

```
structure type buffer_type
  int count;
  item storage[n];
;
var buffer: buffer_type shared;
```

```
Procedure producer(item)
  region buffer when (buffer.count < n) do
  begin
    buffer.storage[in] = item;
    in = in + 1 mod n;
    count = count + 1;
  end
end procedure

Procedure consumer(item)
  region buffer when (buffer.count > 0) do
  begin
    item = buffer.storage[out];
    out = out + 1 mod n;
    count = count - 1;
  end
end procedure
```

An important point to note with the critical section mechanism is the following: consumer process may attempt to enter the critical section region and discover that the condition (buffer.count ¿ 0) actually evaluates to false. As a result, the consumer process cannot enter the critical section and it goes to sleep. However, at some time in the future, when the producer process enters its region and finishes successfully, it is quite possible (and in fact, for this case certain), that the condition (buffer.count ¿ 0) is no longer false for the consumer, and in fact, the consumer can now enter the critical section region. In other words, every time any process exits its region, any other processes' conditions for trying to enter the region must be re-evaluated to determine if any of those processes is now eligible for entering the critical section. This is precisely the **condition** mechanism that is provided in monitors.

1.2.2 Monitors

A monitor is defined by a set of procedures enclosed inside a monitor section. The monitor is guarenteed to satisfy the property that at any time, only one process is *inside* the monitor, i.e. at any time only one process is executing any of the procedures inside the monitor. In addition, a monitor definition also contains variables that can be only accessed from inside the monitor. A monitor definition looks like the following:

```
type monitor-name = monitor
  var var1:
  var var2:
  procedure P1 (...)
  begin
    ....
  end
  ...
  procedure Pn (...)
  begin
    ...
```

```
  end
  begin
    .... Initialization Code
  end
end  monitor
```

In addition, a **condition** type is introduced which has semantics similar to that of a binary semaphore: A condition variable is essentially a binary semaphore initialized to zero. A process can wait on a condition variable in which case it can be put to sleep. A process can also signal a condition variable. When a process signals a condition variable, only one process waiting on that condition is woken up. If there are no processes sleeping on that condition, then the signal operation is simply ignored. Using **monitors**, a deadlock free solution to the dining philosophers problem can be solved as follows:

```
type dining-philosopher =  monitor
  var st: array[0..4] of  (think, hungry, eat)
  var self: array[0..4] of  condition
  procedure test(i: 0..4)
  begin
    if (st[k + 4_5] != eat & st[k + 1_5] != eat & st[k] == hungry)
      begin
        st[k] = think;
        self[k].signal;
      end
  end

  procedure pickup (i: 0..4)
  begin
    st[i] = hungry;
    test(i);
    if (st[i] != eat) then self[i].wait();
  end

  procedure putdown(i: 0..4)
  begin
    st[i] = think;
    test (i + 4_5);
    test (i + 1_5);
  end
```