consuming the item when buffer is empty.

2.5 Classic Problems of Synchronization

Race condition and critical section problem is solved using various methods. In this session some of the examples are discussed here.

2.5.1 Producer-Consumer Problem

- One or more producers are generating some type of data and placing these in a buffer. A single consumer is taking items out of the buffer one at a time. The system is to be constrained to prevent the ovelap of buffer operations. That is only one agent (producer or consumer) may access the buffer at any one time.
- Fig. 2.6 shows the structure of buffer. The producer can generate items and store them in the buffer at its own space. Each time, an index(in) into the buffer is incremented.

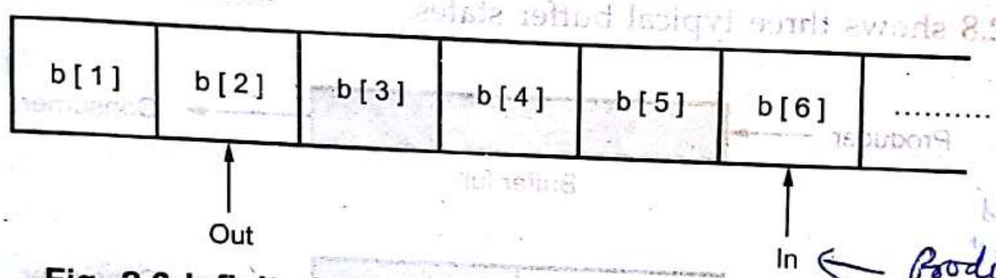


Fig. 2.6 Infinite buffer for producer consumer problem

- The consumer proceeds in a similar fashion but must make sure that it does not attempt to read from an empty buffer.
- Given the infinite buffer, producers may run at any time without restrictions. The buffer itself may be implemented as an array, a linked list, or any other collection of those data items.

Bounded buffer :

- In bounded buffer, producer may produce items only when there are empty buffer slots. A consumer may consume only produced items and must wait when no items are available. All producers must be kept waiting when the buffer is full. When buffer is empty, consumers must wait, for they can never get ahead of producers.

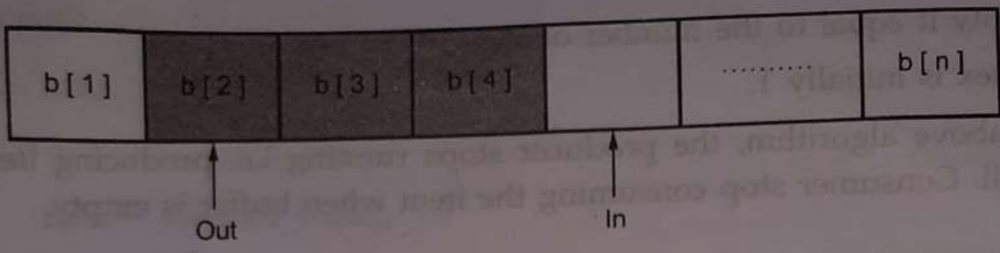


Fig. 2.7 Produced-consumed buffer

- In practice, buffers are usually implemented in a circular fashion. In and out points to the next slot available for a produced item, and to the place where the next item is to be consumed from.
- In real life, the people watch the bin, and it is empty or too full the problem is recoginzed and quickly resolved. However, in a computer system such resolution is not so easy. Consider the case of CPU. The CPU can generate output data much faster than a line printer can print it. Therefore, since this involves a producer and a consumer of two different speeds, we need a buffer where the producer can temporarily store data that can be retrieved by the consumer at a more appropriate speed.
- Fig. 2.8 shows three typical buffer states.

Solution :- Creating 2 Counting Semaphores "Full" and "empty" to keep track of the current number of full and empty buffers.

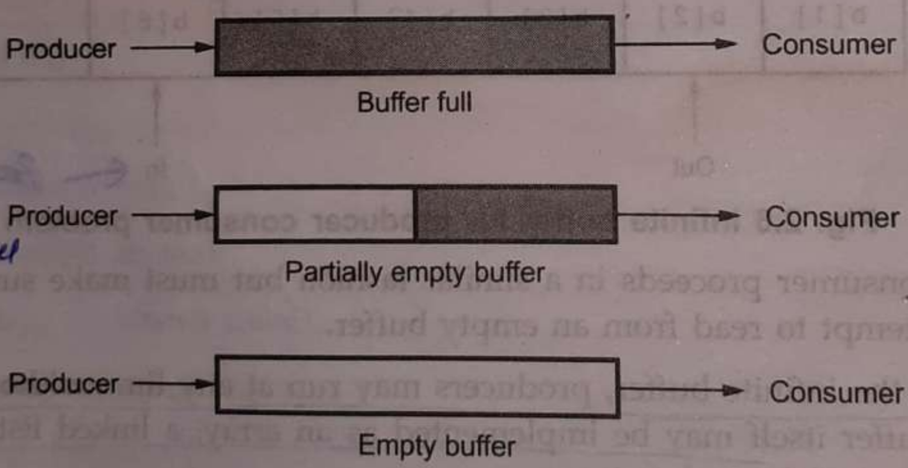


Fig. 2.8 Buffer states

- A solution to the producer-consumer problem satisfy the following conditions.

1. A producer must not overwrite a full buffer.
2. A consumer must not consume an empty buffer.
3. Producers and consumers must access buffers in a mutually exclusive manner.

2.5.2 Readers and Writers Problem

- Reader-writer problem is good example of process synchronization and concurrency mechanisms. It is defined as follows. There is a data area shared among a number of processes. The data area could be a file, a block of main memory etc. There are a number of processes that only read the data area - readers. Processes that only write to the data area - writers. The following conditions must be satisfied.

1. Any number of readers may simultaneously read the file.
2. Only one writer at a time may write to the file.
3. If a writer is writing to the file, no reader may read it.

- Structure of reader process is given below.

```
wait (mutex);
      read count++;
if (readcount==1)
wait (wrt);
signal(mutex);
--------
--------
reading is performed.
---------
---------
wait(mutex);
readcount--;
if(readcount==0)
      signal(wrt);
      signal(mutex);
```

The structure of a writer process is as follows.

```
wait(wrt);
-------
-------
writing is performed
-------
-------
signal(wrt);
```

- The readers-writers problem has several variations, all involving priorities, may be the readers having highest priority or writers having high priority.

- Can the producer-consumer problem be simply a special case of the readers-writers problem with a single writer (the producer) and a single reader (the consumer). **The answer is no.** The producer is not just a writer. It must read queue pointers to determine where to write the next item and it must determine if the buffer is full. Similarly, the consumer is not just a reader because it must adjust the queue pointers to show that it has removed a unit from the buffer.

2.5.3 The Dining Philosophers Problem

- "Five philosophers sit around a circular table. Each philosopher spends his life alternatively thinking and eating. In the center of the table is a large plate of food. A philosopher needs two forks to eat a helping of food. Unfortunately, as philosophy is not as well paid as computing, the philosphers can only afford five forks. One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left."
- The problem is to design a set of processes for philosophers such that each philosopher can eat periodically and none dies of hunger. A philosopher to the left or right of a dining philosopher cannot eat while the dining philosopher is eating, since forks are a shared resource. Fig. 2.9 shows the situation of the dining philosophers.

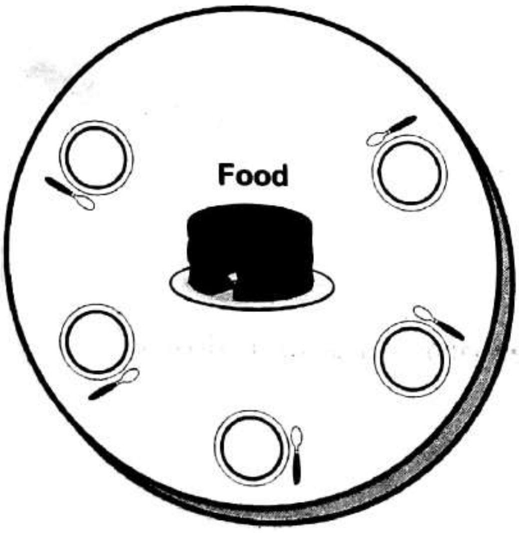
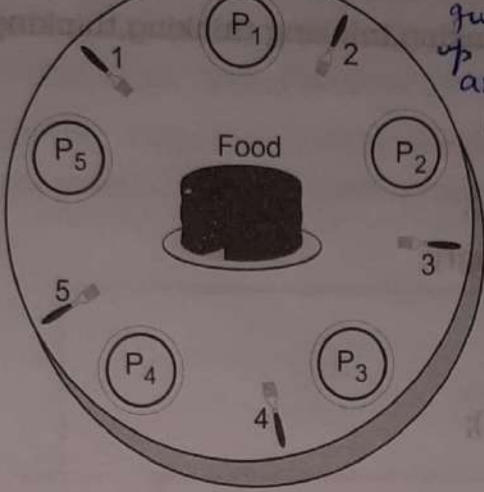


Fig. 2.9 Dining philosophers arrangement

- There are five philosopher processes numbered 1 through 5. Between each pair of philosophers is a fork. Fork is also numbered 1 through 5. So that fork number 3 is between philosophers 2 and 3. This is shown in the Fig. 2.10.



Solution - Another approach is to guarantee that a philosopher can only pick up both fork or none by introducing an arbitrator e.g. a waiter. in order to pick up the forks, a philosopher must ask permission of the waiter. The waiter gives permission to only one philosopher at a time. The waiter can be implemented as mutex. but this approach ~~reduced~~ result in reduced parallelism. if a one philosopher is eating, all other philosopher must wait.

Fig. 2.10 Dining philosophers with fork number

- Each philosopher alternates between thinking and eating. Solution to this problem using semaphore is shown below. Each philosopher picks up his right fork before he tried to pick up his left fork. What happens if the timing works out such that all the philosophers get hungry at the same time, and they all pick up their right forks before any of them gets a chance to try for his left fork ? Then each philosopher i will be holding fork i and waiting for fork i + 1 and they will all wait forever which is shown in Fig. 2.11.

Solution - Such solution is to restrict the no. of philosophers allowed access to the table. if there are N chopsticks but only N-1 philosophers allowed to complete for them, at least one will succeed.

other solution - assign no.s to the resources or forks (ϕ - 5) and each philosopher will always pick up lower-numbered fork in this case, if four of the five philosopher simultaneously pick up their lower numbered fork, only the highest-numbered fork will remain on the table, so the ~~fifth~~ fifth philosopher will not be able to pick up any fork. Moreover only one philosopher will have access to that highest-numbered fork, so they will be able to eat using two fork

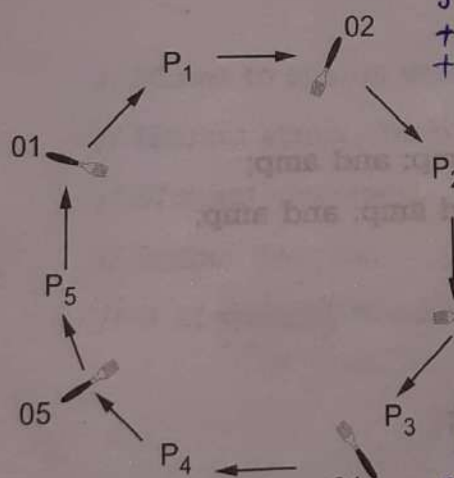


Fig. 2.11 Philosophers pick up fork

- **Following code give the solution for dining philosophers using semaphores.**

```
Semaphore eat [5] = {eating,eating,eating,eating};
Semaphore mutex = 1;
int thinking, hungry, eating;
thinking    = 0;
hungry = 1;
```

~~P1~~ P1 fork1
P2 fork2
P3 fork3
P4 fork4
P5 fork5

fork 5 is allocated to any philosopher who have already 1 fork.

```
eating = 2;
int state [5] ={thinking,thinking,tninking,thinking,thinking};
void takeforks (int j)
      {
            mutex.down ( );
            state [j] = hungry;
            test [j];
            mutex.up ( );
            eat [j].down ( );
      }
 void putforks (int j)
      {
            mutex.down ( );
            state j = thinking;
      test (j == 0 ? 5 : j-1);
      test (j == 4 ? 0 : j+1);

            mutex.up ( );
      }
  void test (int i)
  {
  if (state [j] == hunger and amp; and amp;
        state [j-1]! = eating and amp; and amp;
        state [j+1]! = eating)
  {
            state [j] = eating;
                  eat [j].up ( );
      }
  }
```