CHAPTER 3. MUTUAL EXCLUSION: SOFTWARE SOLUTIONS

case 1: \( p_1 \) read \( \text{wantCS2} \) as false

If \( \text{wantCS2} \) is false, then for \( p_2 \) to enter the critical section it would have to set \( \text{wantCS2} \) to true. From this case, we get the following order of events: \( p_1 \) reads \( \text{wantCS2} \) as false \(< p_2 \) sets the value of \( \text{wantCS2} \) as true.

This order of events implies that \( p_2 \) would set \( \text{turn} = 1 \) before checking the entry condition and after the event of \( p_1 \) reading \( \text{wantCS2} \). On the other hand, \( p_1 \) set \( \text{turn} = 2 \) before reading \( \text{wantCS2} \). Therefore, we have the following order of events in time: \( p_1 \) sets \( \text{turn} = 2 \) \(< p_2 \) reads \( \text{wantCS2} \) as false \(< \text{p2 sets wantCS2 as true} \) \(< p_2 \) sets \( \text{turn} = 1 \) \(< p_2 \) reads \( \text{turn} \). Clearly, \( \text{turn} \) can only be 1 when \( p_2 \) reads it.

Now let us look at what values of \( \text{wantCS1} \) \( p_2 \) can possibly read. Because \( p_1 \) sets \( \text{wantCS1} \) before reading \( \text{wantCS2} \), and \( p_2 \) reads \( \text{wantCS1} \) after writing the value of \( \text{wantCS2} \), we get that \( p_2 \) reads \( \text{wantCS1} \) after \( p_1 \) has set it to true. Therefore, \( p_2 \) can only read \( \text{wantCS1} \) as true.

Because \( p_2 \) reads \( \text{turn} \) as 1 and \( \text{wantCS1} \) as true, it cannot enter the critical section.

case 2: \( p_1 \) read \( \text{turn} \) as 1.

This implies the following order: \( p_2 \) sets \( \text{turn} = 1 \) between \( p_1 \) setting \( \text{turn} = 2 \) and \( p_1 \) reading the value of \( \text{turn} \). Since \( p_2 \) reads the value of \( \text{turn} \) only after setting \( \text{turn} = 1 \), we know that it can only read it as 1. Also \( \text{wantCS1} \) is set before \( p_1 \) set \( \text{turn} = 2 \). Therefore, \( p_1 \) wrote \( \text{wantCS1} \) before \( p_2 \) set \( \text{turn} = 1 \). This implies that \( p_2 \) can only read the value of \( \text{wantCS1} \) as true. Therefore, \( p_2 \) cannot enter the critical section.

It is easy to see that the algorithm satisfies the progress property. If both the processes are forever checking the entry protocol in the while loop, then we get

\[ \text{wantCS2} \land (\text{turn} = 2) \land \text{wantCS1} \land (\text{turn} = 1) \]

which is clearly false because \( (\text{turn} = 2) \land (\text{turn} = 1) \) is false.

The proof of freedom from starvation is left as an exercise. The reader can also verify that Peterson’s algorithm does not require strict alternation of the critical sections—a process can repeatedly use the critical section if the other process is not interested in it.

3.1.2 Mutual Exclusion for \( N \) Processes

Although Peterson’s algorithm satisfies all the properties that we initially required from the protocol, it suffers from the following disadvantages.

1. It works only for two processes. Although the algorithm can be extended to \( N \) processes by repeated invocation of the entry protocol, the resulting algorithm is more complex.

2. Peterson’s algorithm uses variables that can be written by multiple writers. A solution in which there is at most a single writer to any shared variable is more desirable. The correctness of Peterson’s algorithm depends on the fact that concurrent writes to the turn variable results in a valid value.

We now describe Lamport’s bakery algorithm that overcomes these two disadvantages. The algorithm is similar to that used by bakeries in serving customers. Each customer that arrives at bakery receives a number. The server serves customer with the smallest number.

In a concurrent environment, it is difficult to ensure that every process gets a unique number. So in case of a tie, we use process ids to choose the smaller process.

The algorithm requires a process \( p_i \) to go through two main steps before it can enter the critical section. In the first step, it is required to choose a number. To do that, it reads numbers of all other processes and chooses its number as one bigger than the maximum number it read. We will call this step as \( \text{doorway} \). In the second step the process \( p_i \) checks if it can enter the critical section as follows. For every other process \( p_j \), \( p_i \) first checks whether \( p_j \) is currently in the doorway. If \( p_j \) is in the doorway, then \( p_i \) waits for \( p_j \) to get out of the doorway. Then, \( p_i \) waits for \( \text{number}[j] \) to be 0 or \( \text{number}[i], i < \text{number}[j], j \). When \( p_i \) is successful in verifying above condition for all other processes, it can enter the critical section.

```java
boolean choosing[n]; // initially false
int Number[n]; // initially false
```
3.1. MUTUAL EXCLUSION

// first choose a number
choosing[i] = true;
for (int j = 0; j < N; j++)
  if (number[j] > number[i])
    number[i] = number[j];
choosing[i] = false;

for (int j = 0; j < N; j++) {
  while (choosing[j]) ;
  while (number[j] &&
    ((number[j] < number[i]) ||
    ((number[j] == number[i]) && j < i))) ;
}
criticalSection();

number[i] = 0; // exit protocol
nonCriticalSection();

We first prove the assertion:

(A1) If a process pi is in critical section and some other process pk has already chosen its number, then (number[i],i) < number[k],k).

If the process pi is in critical section, then it managed to get out of the kth iteration of the for loop. This implies that either (number[k] = 0) or (number[i],i) < number[k],k) at that iteration. First assume that process pi has read number[k] as 0. This means that process pk must not have finished choosing the number yet. There are two cases. Either pk has not entered the doorway or it has entered the doorway but not exited yet. If pk has not entered the doorway, it will read the latest value of number[i] and is guaranteed to have number[k] > number[i]. If it had entered the doorway, this entry must be after pi had checked choosing[k] because pi waits for pk to finish choosing before checking the condition (number[k] = 0) ∨ ((number[i],i) < number[k],k)). This again means that that pk will read the latest value of number[i] and therefore (number[i] < number[k]).

If (number[i],i) < number[k],k) at kth iteration, then this will continue to hold because number[i] does not change and number[k] can only increase.

We can also claim the assertion:

(A2) If a process pi is in critical section, then (number[i] > 0).

(A2) is true because from the text it is clear that the value of any number is at least 0 and a process executes increment operation on its number before entering the critical section.

Now showing that the bakery algorithm satisfies mutual exclusion is trivial. If two processes pi and pk are in critical section, then from (A2) we know that both of their numbers are nonzero. From (A1) it follows that (number[i],i) < number[k],k) and vice versa, which is a contradiction.

The algorithm also satisfies starvation freedom because any process that is waiting to enter the critical section will eventually have the smallest nonzero number. This process will then succeed in entering the critical section.