3.1 Following is Dekker's algorithm, the first solution to the critical section problem for two processes:

```
bool enter1 = false, enter2 = false;
int turn = 1;

process P1 {
  while (true) {
    enter1 = true;
    while (enter2)
      if (turn == 2) {
        enter1 = false;
        while (turn == 2) skip;
        enter1 = true;
      }
    critical section;
    enter1 = false; turn = 2;
    noncritical section;
  }
}

process P2 {
  while (true) {
    enter2 = true;
    while (enter1)
      if (turn == 1) {
        enter2 = false;
        while (turn == 1) skip;
        enter2 = true;
      }
        critical section;
        enter2 = false; turn = 1;
        noncritical section;
      }
    }
```

Explain clearly how the program ensures mutual exclusion, avoids deadlock, avoids unnecessary delay, and ensures eventual entry. For the eventual entry property, how many times can one process that wants to enter its critical section be bypassed by the other before the first gets in? Explain.

3.4 Suppose a computer has an atomic Compare-and-Swap instruction, which has the following effect:

```
CSW(a, b, c):
  ⟨ if (a == c)
      { c = b; return (0); }
    else
      { a = c; return (1); } ⟩
```

Parameters **a**, **b**, and **c** are simple variables, such as integers. Using **CSW**, develop a solution to the critical section problem for **n** processes. Do not worry about the eventual entry property. Describe clearly how your solution works and why it is correct.

3.9 Consider the following variation on the `n`-process tie-breaker algorithm [Block and Woo 1990]:

```
int in = 0, last[1:n];   # shared variables
process CS[i = 1 to n] {
  int stage;
  while (true) {
    ⟨in = in + 1;⟩; stage = 1; last[stage] = i;
    ⟨await (last[stage] != i or in <= stage);⟩
    while (last[stage] != i) {  # go to next stage
      stage = stage + 1; last[stage] = i;
      ⟨await (last[stage] != i or in <= stage);⟩
    }
    critical section;
    ⟨in = in - 1;⟩
    noncritical section;
  }
}
```

(a) Explain clearly how this program ensures mutual exclusion, avoids deadlock, and ensures eventual entry.

(b) Compare the performance of this algorithm to that of the tie-breaker algorithm in Figure 3.7. In particular, which is faster if only one process is trying to enter the critical section? How much faster? Which is faster if all `n` processes are trying to enter the critical section? How much faster?

(c) Convert the coarse-grained solution above to a fine-grained solution in which the only atomic actions are reading and writing variables. Do not assume increment and decrement are atomic. (*Hint*: Change `in` to an array.)

3.8 Suppose your machine has the following atomic instruction:

```
flip(lock)
  ⟨ lock = (lock + 1 ) % 2;   # flip the lock
    return (lock); ⟩          # return the new value
```

Someone suggests the following solution to the critical section problem for *two* processes:

```
int lock = 0;   # shared variable

process CS[i = 1 to 2] {
  while (true) {
    while (flip(lock) != 1)
      while (lock != 0) skip;
    critical section;
    lock = 0;
    noncritical section;
  }
}
```

(a) Explain why this solution will *not* work—in other words, give an execution order that results in both processes being in their critical sections at the same time.

(b) Suppose that the first line in the body of `flip` is changed to do addition module `3` rather than modulo `2`. Will the solution now work for two processes? Explain your answer.

3.11 In the bakery algorithm (Figure 3.11), the values of `turn` are unbounded if there is always at least one process in its critical section. Assume reads and writes are atomic. Is it possible to modify the algorithm so that values of `turn` are always bounded? If so, give a modified algorithm. If not, explain why not.

```
int turn[1:n] = ([n] 0);
process CS[i = 1 to n] {
  while (true) {
    turn[i] = 1; turn[i] = max(turn[1:n]) + 1;
    for [j = 1 to n st j != i]
      while (turn[j] != 0 and
                 (turn[i],i) > (turn[j],j)) skip;
    critical section;
    turn[i] = 0;
    noncritical section;
  }
}
```

Figure 3.11 Bakery algorithm: Fine-grained solution.

3.16 Consider the following implementation of a single n-process barrier:

```
int arrive[1:n] = ([n] 0);   # shared array
code executed by Worker[1]:
  arrive[1] = 1;
  ⟨await (arrive[n] == 1);⟩
code executed by Worker[i = 2 to n]:
  ⟨await (arrive[i-1] == 1);⟩
  arrive[i] = 1;
  ⟨await (arrive[n] == 1);⟩
```

(a) Explain how this barrier works.

(b) What is the time complexity of the barrier?

(c) Extend the above code so that the barrier is reusable.

3.17 Assume there are `n` worker processes, numbered from `1` to `n`. Also assume that your machine has an atomic increment instruction. Consider the following code for an `n`-process barrier that is supposed to be reusable:

```
int count = 0; go = 0;   # shared variables
code executed by Worker[1]:
  ⟨await (count == n-1);⟩
  count = 0;
  go = 1;
code executed by Worker[2:n]:
  ⟨count++;⟩
  ⟨await (go == 1);⟩
```

(a) Explain what is wrong with the above code?

(b) Fix the code so that it works. Do not use any more shared variables, but you may introduce local variables.

(c) Suppose the above code were correct. Assume all processes arrive at the barrier at the same time. How long does it take before every process can leave the barrier? Count each assignment statement as `1` time unit, and count each **`await`** statement as `1` time unit once the condition becomes true.

3.19 Modify each of the following algorithms to use only **k** processes instead of **n** processes. Assume that **n** is a multiple of **k**.

(a) The parallel prefix computation in Figure 3.17.

(b) The linked-list computation in Figure 3.18.

(c) The grid computation in Figure 3.19.

3.21 Assume there are **n** processes **P[1:n]** and that **P[1]** has some local value **v** that it wants to broadcast to all the others. In particular, the goal is to store **v** in every entry of array **a[1:n]**. The obvious sequential algorithm requires linear time. Write a data parallel algorithm to store **v** in **a** in logarithmic time.

3.26 The following region-labeling problem arises in image processing. Given is integer array **image[1:n,1:n]**. The value of each entry is the intensity of a pixel. The neighbors of a pixel are the four pixels to the left, right, above, and below it. Two pixels belong to the same region if they are neighbors and they have the same value. Thus, a region is a maximal set of pixels that are connected and that all have the same value.

The problem is to find all regions and assign every pixel in each region a unique label. In particular, let **label[1:n,1:n]** be a second matrix, and assume that the initial value of **label[i,j]** is **n*i + j**. The final value of **label[i,j]** is to be the largest of the initial labels in the region to which pixel **[i,j]** belongs.

Write a data parallel grid computation to compute the final values of **label**. The computation should terminate when no **label** changes value.