1. Consider the following operations on two shared variables x and y. Two processes work on them.

shared int x = 10000;
shared int y = 5000;

P1	P2

`y -= 1000`
`x += 2000`
`y -= 2000`
	`y += 4000`

What are the possible values of x and y after P1 and P2 complete?

x = 12000 always.
y can be 6000, 7000, 8000 or 9000

2. Let *x* and *y* be shared int variables. We want to increment *x* atomically by *y*. We make the following implementation based on the hardware-level compare-and-swap instruction. Will it always work correctly?

```
increment ( shared int *x, shared int y )
{
    int temp;

    do {
        temp = *x;
    } while (compare_and_swap(x, temp, temp + y) != temp);
}
```

No. Since *y* is a shared variable, it may change after a process reads it but before it is able to call `compare_and_swap` with `temp + y` as the third argument.

What is the solution? A lock is a possibility.

3. [*Parallel-sum problem*]

Let A be an aray of size $n = 2^t$ and one-based indexing. Assume that there are n processors, and that the i-th processor updates the value of A[i] using the following algorithm. The final sum is computed in A[n].

```
for j = 1, 2, . . . , t, do:
        for i = 1, 2, . . . , n, do:
                if (i % 2^j == 0), then A[i] += A[i – 2^(j – 1)];
```

Explain the need for synchronization for this problem. Use semaphores to synchronize.

```
shared int A[n];
semaphore s[n] = {1, 1, . . . , 1};

for j = 1, 2, . . . , t, do:
        for i = 1, 2, . . . , n, do:
                if ( i % 2^j == 0), then:
                        wait(s[i]);
                        A[i] = A[i] + A[i – 2^(j – 1)];
                        if (i + 2^j < n) signal(s[i + 2^j]);
```

This solution sends some redundant (although harmless) signals in the last line. To avoid these, you can add another condition $((i + 2^j)\ \%\ 2^{j+1} == 0)$.

4. [*Designated-reader and writer problem*]

Consider the reader-writer problem with designated readers (that is, each item is meant for a specific reader). There are n reader processes, where n is known beforehand. There are one or more writer processes. Items are stored in a buffer of unlimited capacity. Every item is written by a writer, and is designated for a particular reader. Solve this problem so that no process makes any busy wait.

```
semaphore rw_mutex = 1;
semaphore r_mutex[n] = {0, 0, . . . , 0};

reader (i)
{
        wait(r_mutex[i]);
        while (true) {
                wait(rw_mutex);
                Read and remove one item from buffer, that is meant for the i-th reader;
                signal(rw_mutex);
                wait(r_mutex[i]);
        }
}

writer ()
{
        while (true) {
                Generate item for reader i;
                wait(rw_mutex);
                Write (item, i) to buffer;
                signal(rw_mutex);
                signal(r_mutex[i]);
        }
}
```

5. **[*Starvation-free reader-writer problem*]**

Implement under the assumption that the semaphore queues are FIFO queues.

```
shared int read_count = 0;
semaphore rw_mutex = 1;
semaphore r_mutex = 1;
semaphore q_mutex = 1;

reader ()
{
        wait(q_mutex);
        wait(r_mutex);
        ++read_count;
        if (read_count == 1) wait(rw_mutex);
        signal(q_mutex);
        signal(r_mutex);

        read();

        wait(r_mutex);
        – –read_count();
        if (read_count == 0) signal(rw_mutex);
        signal(r_mutex);
}

writer ()
{
        wait(q_mutex);
        wait(rw_mutex);
        signal(q_mutex);

        write();

        signal(rw_mutex);
}
```

6. [Sleeping barber problem]

```
shared light_in_the_waiting_room = green;
shared chairs_in_the_waiting_room = all_empty;

barber ()
{
        while (true) {
                inspect the waiting room;
                if (there are no customers), then
                        set the status light of waiting room to green (available);
                        sleep until woken up by a customer;
                set the status light of waiting room to red (busy);
                serve the next customer;
        }
}

customer ()
{
        enter the waiting room;
        if the status light is red {
                if all of the n chairs in the waiting room are occupied, then leave;
                occupy an empty chair in the waiting room;
                sleep until woken up by the barber;
        }
        enter barber's room;
        wake up the barber if sleeping;
        have hair-cut and leave;
}
```

(a) Where are race conditions possible?

(i) For occupying empty chairs

(ii) Barber sleeping. Two (or more) new customers come at the same time. Both see the status light green and enter barber's room.

(iii) Barber finishes a hair-cut, inspects the waiting room, finds nobody. Barber is preempted. A new customer comes, sees the red light, and sleeps. Barber is rescheduled, sets the status light to green, and sleeps.

(b) Solve using semaphores.

```
shared int no_of_empty_chairs = n;

semaphore barber_mtx = 0;          /* Barber waits on the mutex *
semaphore customer_mtx = 0;        /* Customers wait on this mutex */
semaphore chair_mtx = 1;           /* Mutex to protect no_of_empty_chairs */

barber ()
{
        while (true) {
                wait(barber_mtx);
                wait(chair_mtx);
                ++no_of_empty_chairs;
                signal(customer_mtx);
                signal(chair_mtx);

                hair_cut();
        }
}

customer ()
{
        wait(chair_mtx);
        if (no_of_empty_chairs == 0) {
                signal(chair_mtx);
        } else {
                – –no_of_empty_chairs;
                signal(barber_mtx);
                signal(chair_mtx);
                wait(customer_mtx);

                have_hair_cut();
        }
}
```