The Readers and Writers Problem. Another famous problem is the readers and writers problem, which models access to a database. Assume that several processes want to write to or read from the shared database information. It is allowed to read simultaneously from database but if one process is writing to the database no other processes including the reading processes may have access to the database. The question is how to program the readers and writers?

One of the solutions assumes that first reader that accesses the database decrements a semaphore type variable \(db\) to forbid entering to critical region the writers. Other readers increment counter \(rc\) to form a queue for accessing the database. The last reader increments the variable \(bd\), allowing the blocked writer, if any, to get in. If a writer appears while several readers are in the database, the writer must wait. If new readers appear and there is at least one reader in the database, the writer must keep waiting until no more readers are interested in the database.

To prevent this situation, the program could be written slightly differently, when a reader arrives and a writer is waiting, the reader is suspended behind the writer instead of being admitted immediately. In this way, a writer has to wait for readers that were active when it arrived to finish but does not have to wait for readers that came along after it. The disadvantage of this solution is that it achieves less concurrency and thus lower performance.

typedef int semaphore; //use your imagination
semaphore mutex=1; //controls access to 'rc'
semaphore db=1; //controls access to the database
int rc=0; // #of processes reading or wanting to

void reader(void)
{
    while(TRUE){ //repeat forever
        down(&mutex); //get exclusive access to 'rc'
        rc=rc+1; //one reader more now
        if(rc==1) down(&db); //if this is the first reader ...
        up(&mutex); //release exclusive access to 'rc'
        read_data_base(); //access the data
        down(&mutex); //get exclusive access to 'rc'
        rc=rc-1; //one reader fewer now
        if(rc==0) up(&db); //if this is the last reader ...
        up(&mutex); //release exclusive access to 'rc'
        use_data_read(); //noncritical region
    }
}
void writer(void)
{
    while(TRUE){ //repeat forever
        think_up_data(); //noncritical region
        down(&db); //get exclusive access
        write_data_base(); //update the data
        up(&db); //release exclusive access
    }
}

The Sleeping Barber Problem. The barber shop has one barber, one barber chair, and \( n \) chairs for waiting customers, if any. If there is no customer, the barber sits down in the barber chair and falls asleep. When a customer arrives, he has to wake up the sleeping barber. If additional customers arrive while the barber is cutting a customer’s hair, they either sit down, if there are empty chairs, or leave the shop, if all chairs are full. The problem is to program the barber and the customers without having race conditions.

The solution is based on three semaphores: customers, which counts the customers; barbers, which represents the number of idle barbers (0 or 1); and mutex, which is used for mutual exclusion. We also need a variable, waiting, which also counts the waiting customers. It is essentially a copy of customers. The reason for having waiting is that there no way to read the current value of a semaphore, and in this solution, a customer entering the barber shop has to count the number of waiting customers. If it is less than the number of chairs, he stays, otherwise, he leaves.

When the first customer arrives, he executes customer, starting by acquiring mutex to enter a critical section. If another customer enters shortly thereafter. The second one will not be able to do anything until the first one has released mutex. The customer then checks to see if the number of waiting customers is less than the number of chairs. If not, he releases mutex and leaves without haircut. If there is an available chair, the customer increments the integer variable, waiting. Then it ups the semaphore customers, thus waking up the barber.
Exercise. Suppose that there are 1 barber chair, and 3 customer chairs in a barber’s shop. Suppose that the processes are called in the order:

Barber, Customer1, Customer2, Customer3, Customer4, Customer5, Barber, Customer1, Barber, Customer2.

Trace the algorithm, and find the values of the variables after each step has been completed.

Solution.
In LINUX, you can type multiple command line for sequential and/or parallel execution. The following is a brief description of the syntax for sequential execution of commands specified in one command line. In the following session, the date and echo commands execute sequentially as separate processes.

$ date; echo Hello!

You can specify parallel execution of commands in a command line by ending each command with an ampersand (&). The commands which terminate with & also execute in the background. No spaces are required before or after an &, but use can use spaces for clarity. In the following session, the commands date and echo commands execute in parallel, followed by sequential execution of the uname and who commands. The date and echo commands execute in the background, and the uname and who commands execute in the foreground.

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
<th>customers</th>
<th>barbers</th>
<th>mutex</th>
<th>Waiting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initially</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Barber</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Customer1</td>
<td>1</td>
<td>0</td>
<td>0, 1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Customer2</td>
<td>2</td>
<td>0</td>
<td>0, 1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Customer3</td>
<td>3</td>
<td>0</td>
<td>0, 1</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>Customer4</td>
<td>3</td>
<td>0</td>
<td>0, 1</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>Customer5</td>
<td>3</td>
<td>0</td>
<td>0, 1</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>Barber</td>
<td>2</td>
<td>1</td>
<td>0, 1</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>Customer1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>Barber</td>
<td>1</td>
<td>1</td>
<td>0, 1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>Customer2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>