The Sleeping Barber Problem
Statement of the Problem and a Solution

From Tanenbaum (Second Edition), pages 129 to 132

Another classical IPC problem takes place in a barber shop. The barber shop has one barber, one barber chair, and *n* chairs for waiting customers, if any, to sit on. If there are no customers present, the barber sits down in the barber chair and falls asleep, as illustrated in [Figure 2-35]. 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 getting into race conditions. This problem is similar to various queueing situations, such as a multiperson helpdesk with a computerized call waiting system for holding a limited number of incoming calls.

Our solution uses three semaphores, *customers*, which counts waiting customers (excluding the customer in the barber chair, who is not waiting), *barbers*, the number of barbers (0 or 1) who are idle, waiting for customers, and *mutex*, which is used for mutual exclusion. We also need a variable, *waiting*, which also counts the waiting customers. The reason for having *waiting* is that there is no way to read the current value of a semaphore. In this solution, a customer entering the shop has to count the number of waiting customers. If it is less than the number of chairs, he stays; otherwise, he leaves.

Our solution is shown [below]. When the barber shows up for work in the morning, he executes the procedure *barber*, causing him to block on the semaphore *customers* because it is initially 0. The barber then goes to sleep, as shown in [Figure 2.35]. He stays asleep until the first customer shows up.

When a customer arrives, he executes *customer*, starting by acquiring *mutex* to enter a critical region. If another customer enters shortly thereafter, the second one will no 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 a haircut.

If there is an available chair, the customer increments the integer variable, *waiting*. Then he does an *Up* on the semaphore *customers*, thus waking up the barber. At this point, the customer and the barber are both awake. When the customer releases *mutex*, the barber grabs it, does some housekeeping, and begins the haircut.

When the haircut is over, the customer exits the procedure and leaves the shop. Unlike our earlier examples, there is no loop for the customer because each one gets only one haircut. The barber loops, however, to try to get the next customer. If one is present, a haircut is given. If not, the barber goes to sleep.

As an aside, it is worth pointing out that although the readers and writers and sleeping barber problems do not involve data transfer, they still belong to the area of IPC because they involve synchronization between multiple processes.

```
#define CHAIRS 5                /* # chairs for waiting customers */

typedef int semaphore;          /* use your imagination */

semaphore customers = 0;        /* # of customers waiting for service */
semaphore barbers = 0;          /* # of barbers waiting for customers */
semaphore mutex = 1;            /* for mutual exclusion */
int waiting = 0;                /* customer are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);       /* go to sleep if # of customers is 0 */
        down(&mutex);           /* acquire access to "waiting' */
        waiting = waiting - 1;  /* decrement count of waiting customers */
        up(&barbers);           /* one barber is now ready to cut hair */
        up(&mutex);             /* release 'waiting' */
        cut_hair();             /* cut hair (outside critical region */
    }
}

void customer(void)
{
    down(&mutex);               /* enter critical region */
    if (waiting < CHAIRS) {     /* if there are no free chairs, leave */
        waiting = waiting + 1;  /* increment count of waiting customers */
        up(&customers);         /* wake up barber if necessary */
        up(&mutex);             /* release access to 'waiting' */
        down(&barbers);         /* go to sleep if # of free barbers is 0 */
        get_haircut();          /* be seated and be served */
    } else {
        up(&mutex);             /* shop is full; do not wait */
    }
}
```