1 **The Sleeping-Barber Problem.**

A barbershop consists of a waiting room with *n* chairs and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber.

(1) Write a program to coordinate the barber and the customers.

Answer:

We use 3 semaphores. Semaphore ***customers*** counts waiting customers; semaphore ***barbers*** is the number of idle barbers (0 or 1); and ***mutex*** is used for mutual exclusion. A shared data variable ***customers1*** also counts waiting customers. It is a copy of ***customers***. But we need it here because we can't access the value of semaphores directly. We also need a semaphore ***cutting*** which ensures that the barber won't cut another customer's hair before the previous customer leaves.

```
// shared data
semaphore customers = 0;
semaphore barbers = 0;
semaphore cutting = 0;
semaphore mutex = 1;
int customer1 = 0;

void barber() {
        while(true) {
                wait(customers);        //sleep when there are no waiting customers
                wait(mutex);            //mutex for accessing customers1
                customers1 = customers1 - 1;
                signal(barbers);
                signal(mutex);
                cut_hair();
        }
}

void customer() {
        wait(mutex);            //mutex for accessing customers1
        if (customers1 < n) {
                customers1 = customers1 + 1;
                signal(customers);
                signal(mutex);
                wait(barbers);  //wait for available barbers
                get_haircut();
        }
```

```
        else {   //do nothing (leave) when all chairs are used.
                 signal(mutex);
        }
}
cut_hair(){
        waiting(cutting);
}
get_haircut(){
        get hair cut for some time;
        signal(cutting);
}
```

(2) Consider the Sleeping-Barber Problem with the modification that there are k barbers and k barber chairs in the barber room, instead of just one. Write a program to coordinate the barbers and the customers.

Answer:

```
// shared data
semaphore waiting_room_mutex = 1;
semaphore barber_room_mutex = 1;
semaphore barber_chair_free = k;
semaphore sleepy_barbers = 0;
semaphore barber_chairs[k] = {0, 0, 0, …};
int barber_chair_states[k] = {0, 0, 0, …};
int num_waiting_chairs_free = N;

boolean customer_entry( ) {
        // try to make it into waiting room
        wait(waiting_room_mutex);
        if (num_waiting_chairs_free == 0) {
                signal(waiting_room_mutex);
                return false;
        }

        num_waiting_chairs_free--; // grabbed a chair
        signal(waiting_room_mutex);

        // now, wait until there is a barber chair free
        wait(barber_chair_free);

        // a barber chair is free, so release waiting room chair
        wait(waiting_room_mutex);
        wait(barber_room_mutex);
        num_waiting_chairs_free++;
        signal(waiting_room_mutex);
```

```
        // now grab a barber chair
        int mychair;
        for (int I=0; I<k; I++) {
                if (barber_chair_states[I] == 0) { // 0 = empty chair
                mychair = I;
                break;
                }
        }
        barber_chair_states[mychair] = 1; // 1 = haircut needed
        signal(barber_room_mutex);

        // now wake up barber, and sleep until haircut done
        signal(sleepy_barbers);
        wait(barber_chairs[mychair]);

        // great! haircut is done, let's leave. barber
        // has taken care of the barber_chair_states array.
        signal(barber_chair_free);
        return true;
}

void barber_enters() {
        while(1) {
                // wait for a customer
                wait(sleepy_barbers);
                // find the customer
                wait(barber_room_mutex);
                int mychair;
                for (int I=0; I<k; I++) {
                        if (barber_chair_states[I] == 1) {
                                mychair = I;
                                break;
                        }
                }
                barber_chair_states[mychair] = 2; // 2 = cutting hair
                signal(barber_room_mutex);

                // CUT HAIR HERE
                cut_hair(mychair);

                // now wake up customer
                wait(barber_room_mutex);
                barber_chair_states[mychair] = 0; // 0 = empty chair
                signal(barber_chair[mychair]);
                signal(barber_room_mutex);
```

```
        // all done, we'll loop and sleep again
    }
}
```

2. **The Cigarette-Smokers Problem.** Consider a system with three *smoker* processes and one *agent* process. Each smoker continuously rolls a cigarette and then smokes it. But to roll and smoke a cigarette, the smoker needs three ingredients: tobaccor, paper, and matches. One of the smoker processes has paper, another has tobacco, and the third has matches. The agent has an infinite supply of all three materials. The agent places two of the ingredients on the table. The smoker who has the remaining ingredient then makes and smokes a cigarette, signaling the agent on completion. The agent then puts out another two of the three ingredients, and the cycle repeats. Write a program to synchronize the agent and the smokers.

Answer:

We use 5 semaphores. Semaphore ***smoker_tobacco, smoker_match, smoker_paper, agent*** are binary semaphores; and ***lock*** is used for mutual exclusion.

```
// shared data
Semaphore smoker_match=0;
Semaphore smoker_paper=0;
Semaphore smoker_tobacco=0;
Semaphore agent=0;
Semaphore lock=1;

void agent{
    while(1){
        wait( lock );
        randNum = rand( 1, 3 ); // Pick a random number from 1-3
        if ( randNum == 1 ) {
            // Put tobacco on table
            // Put paper on table
            signal( smoker_match );  // Wake up smoker with match
        } else if ( randNum == 2 ) {
            // Put tobacco on table
            // Put match on table
            signal( smoker_paper );  // Wake up smoker with paper
        } else {
            // Put match on table
            // Put paper on table
            signal( smoker_tobacco ); // Wake up smoker with tobacco
        }
        signal( lock );
        wait( agent );  //  Agent sleeps
```

```
        }
}

void Smoker1{
        while(1){
                wait( smoker_tobacco );  // Sleep right away
                wait( lock );
                // Pick up match
                // Pick up paper
                signal( agent );
                signal( lock );
                // Smoke (but don't inhale).
        }
}

void Smoker2{
        while(1){
                wait( smoker_ match );  // Sleep right away
                wait( lock );
                // Pick up tobacco
                // Pick up paper
                signal( agent );
                signal( lock );
                // Smoke (but don't inhale).
        }
}

void Smoker3{
        while(1){
                wait( smoker_ paper );  // Sleep right away
                wait( lock );
                // Pick up match
                // Pick up tobacco
                signal( agent );
                signal( lock );
                // Smoke (but don't inhale).
        }
}
```

The smoker immediately sleeps. When the agent puts the two items on the table, then the agent will wake up the appropriate smoker. The smoker will then grab the items, and wake the agent. While the smoker is smoking, the agent can place two items on the table, and wake a different smoker (if the items placed aren't the same). The agent sleeps immediately after placing the items out. This is something like the producer-consumer problem except the producer can only produce 1 item (although a choice of 3 kinds of items) at a time.