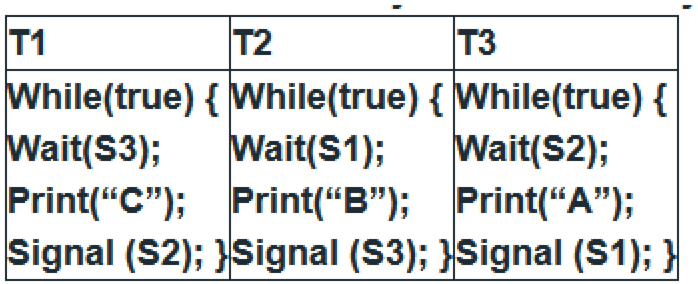
Problem 1

We want to use semaphores to implement a shared critical section (CS) among three processes T1, T2, and T3. We want to enforce the execution in the CS in this order: First T2 must execute in the CS. When it finishes, T1 will then be allowed to enter the CS; and when it finishes T3 will then be allowed to enter the CS; when T3 finishes then T2 will be allowed to enter the CS, and so on, (T2, T1, T3, T2, T1, T3,…).

Write the synchronization solution using a minimum number of binary semaphores and you are allowed to assume the initial value for semaphore variables.

Problem 1



T1	T2	T3
While(true) { Wait(S3); Print(“C”); Signal (S2); }	While(true) { Wait(S1); Print(“B”); Signal (S3); }	While(true) { Wait(S2); Print(“A”); Signal (S1); }

S1=1, S2=0, S3=0

Problem 2

Three concurrent processes X, Y, and Z execute three different code segments that access and update certain shared variables.

Process X executes the P operation (i.e., wait) on semaphores a, b and c;

process Y executes the P operation on semaphores b, c and d;

process Z executes the P operation on semaphores c, d, and a before entering the respective code segments.

After completing the execution of its code segment, each process invokes the V operation (i.e., signal) on its three semaphores.

All semaphores are binary semaphores initialized to **one**.

Which one of the following represents a deadlock-free order of invoking the P operations by the processes?

Option A: X: P(a)P(b)P(c) Y: P(b)P(c)P(d) Z: P(c)P(d)P(a)

Option B: X: P(b)P(a)P(c) Y: P(c)P(b)P(d) Z: P(a)P(c)P(d)

Option C: X: P(a)P(b)P(c) Y: P(c)P(b)P(d) Z: P(c)P(d)P(a)

Option D X: P(b)P(a)P(c) Y: P(b)P(c)P(d) Z: P(a)P(c)P(d)

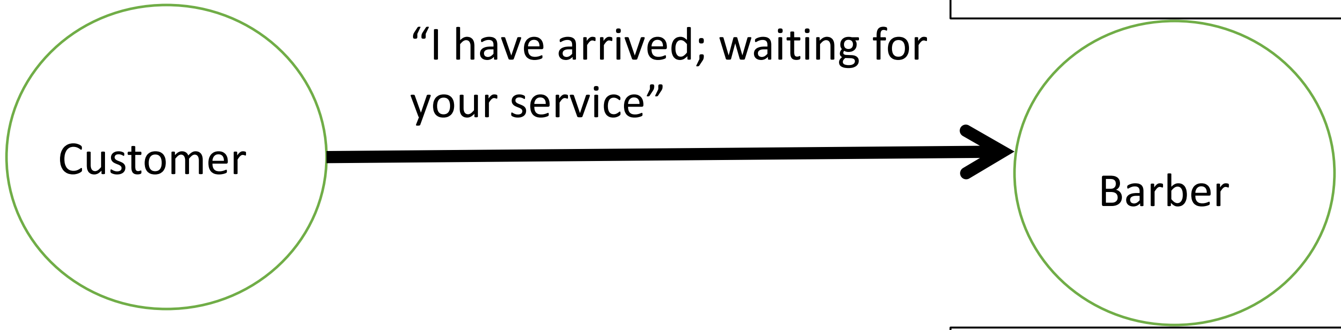
The Sleeping Barber Problem

Challenges

- Actions taken by barber and customer takes unknown amount of time (checking waiting room, entering shop, taking waiting room chair)
- Scenario 1
    - Customer arrives, observe that barber busy
    - Goes to waiting room
    - While he is on the way, barber finishes the haircut
    - Barber checks the waiting room
    - Since no one there, Barber sleeps
    - The customer reaches the waiting room and waits forever
- Scenario 2
    - Two customer arrives at the same time
    - Barber is busy
    - Both customers try to occupy the same chair!

Barber sleeps on “**Customer**”
Customer sleeps on “**Barber**”

One semaphore: customer

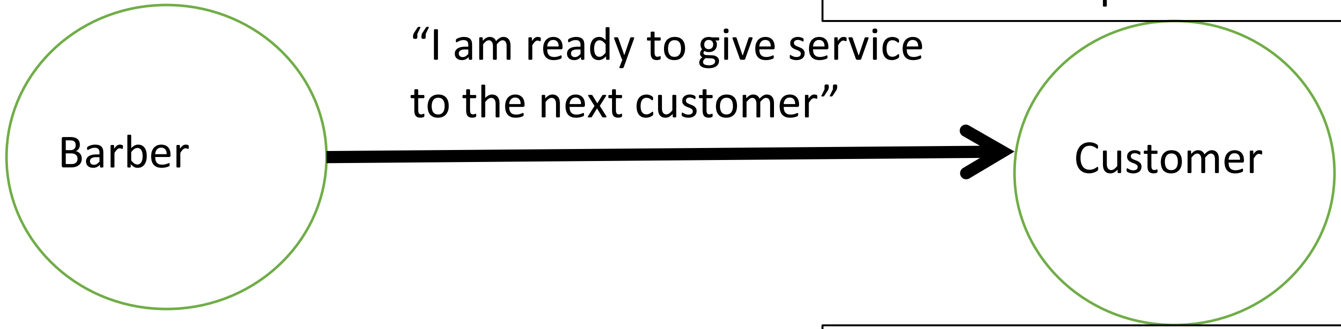


“I have arrived; waiting for your service”

No customer: Barber falls asleep

Barber wakes up, if sleeping

One semaphore: barber

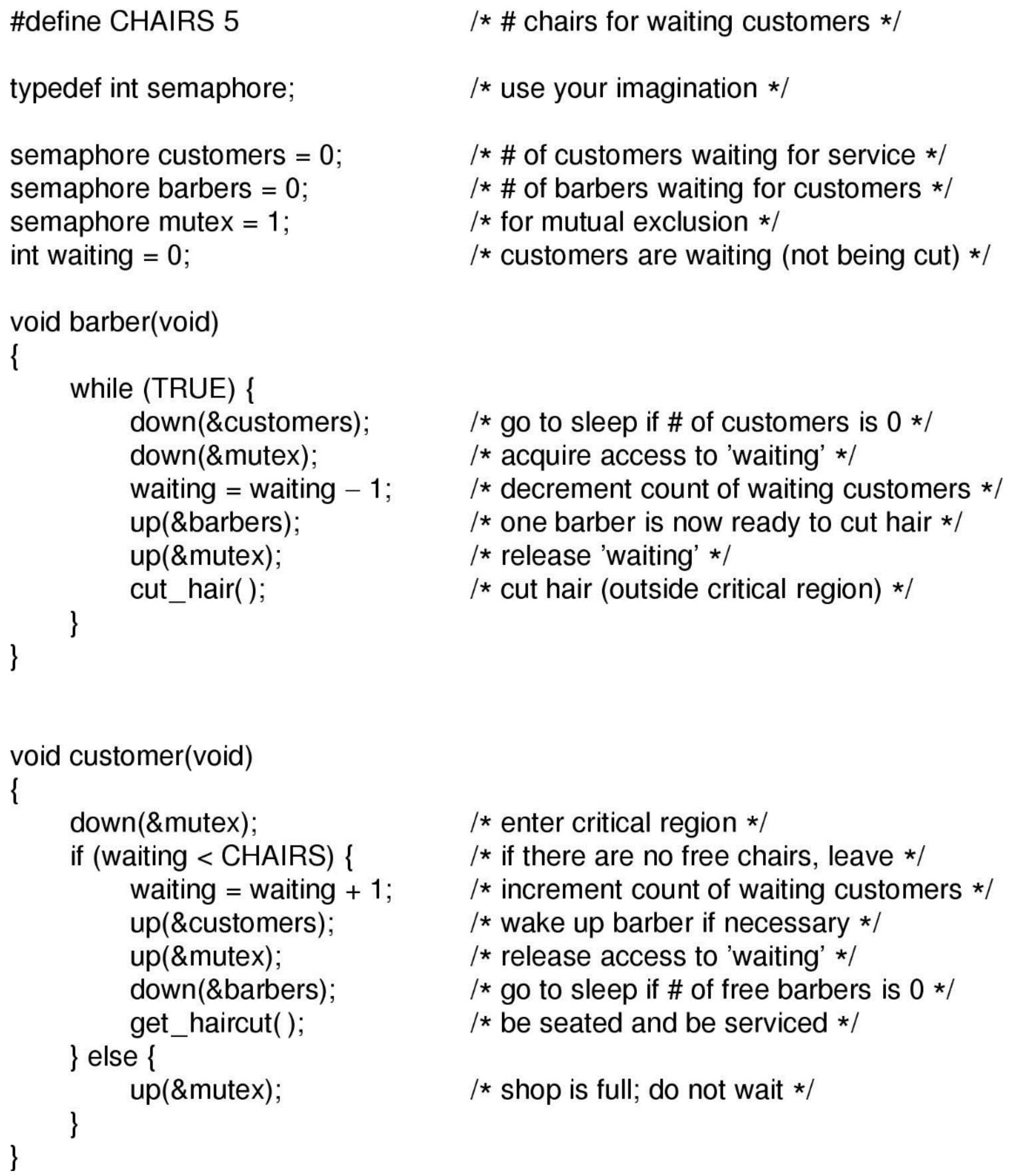


“I am ready to give service to the next customer”

Customer acquires the Barber for service

Customer waits if Barber busy

The Sleeping Barber Problem



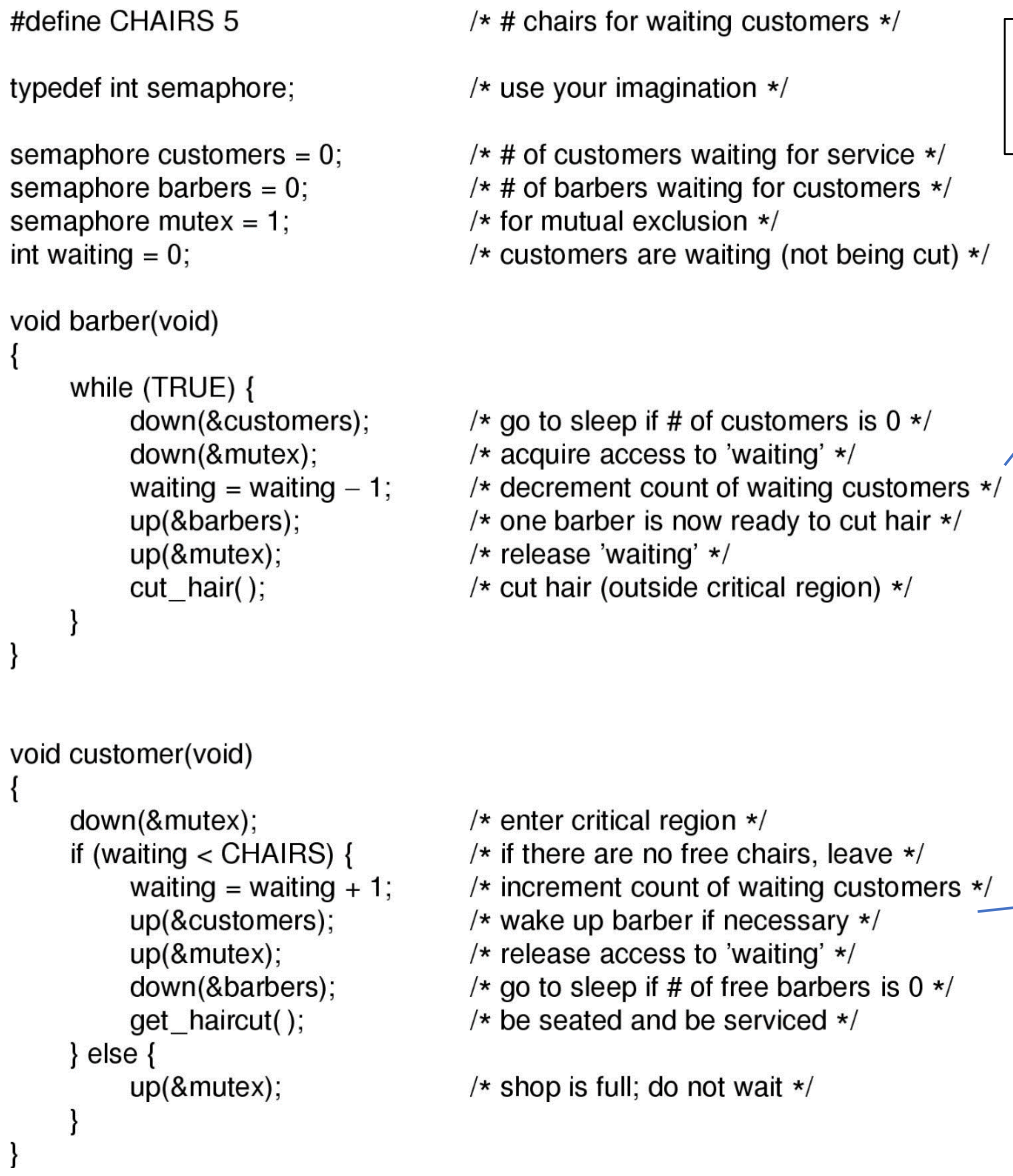
```
#define CHAIRS 5                    /* # chairs for waiting customers */

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;                    /* customers 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 serviced */
     } else {
          up(&mutex);               /* shop is full; do not wait */
     }
}
```

Semaphore Barber: Used to call a waiting customer.
Barber=1: Barber is ready to cut hair and a customer is ready (to get service) too!
Barber=0: customer occupies barber or waits

Semaphore customer: Customer informs barber that "I have arrived; waiting for your service"

Mutex: Ensures that only one of the participants can change state at once

The Sleeping Barber Problem



```
#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;                    /* customers 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 serviced */
     } else {
          up(&mutex);               /* shop is full; do not wait */
     }
}
```

Barber sleeps on “**Customer**”
Customer sleeps on “**Barber**”

For Barber: Checking the waiting room and calling the customer makes the **critical section**

For customer: Checking the waiting room and informing the barber makes its **critical section**

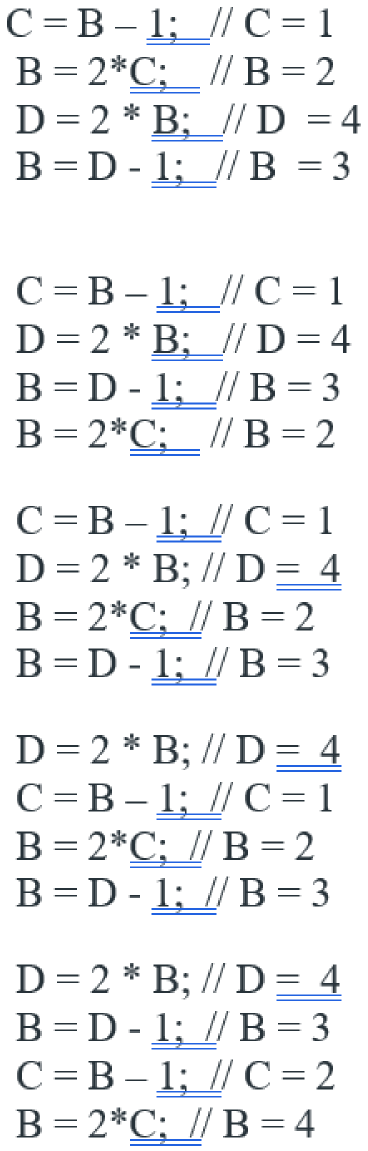
Problem 3

- The following two functions P1 and P2 that share a variable B with an initial value of 2 execute concurrently.

```
P1()
{
  C = B – 1;
  B = 2*C;
}

P2()
{
  D = 2 * B;
  B = D - 1;
}
```

The number of distinct values that B can possibly take after the execution

Problem 3



```
C = B – 1;   // C = 1
B = 2*C;    // B = 2
D = 2 * B;   // D  = 4
B = D - 1;   // B  = 3


C = B – 1;   // C = 1
D = 2 * B;   // D = 4
B = D - 1;   // B = 3
B = 2*C;    // B = 2

C = B – 1;  // C = 1
D = 2 * B; // D =  4
B = 2*C;  // B = 2
B = D - 1;  // B = 3

D = 2 * B; // D =  4
C = B – 1;  // C = 1
B = 2*C;  // B = 2
B = D - 1;  // B = 3

D = 2 * B; // D =  4
B = D - 1;  // B = 3
C = B – 1;  // C = 2
B = 2*C;  // B = 4
```

Problem 4

Consider the reader-writer problem with designated readers. There are n reader processes, where n is known beforehand. There are one or more writer processes. Items are stored in a buffer. Every item is written by a writer and is designated for a particular reader.

```
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]);
        }
}
```