Semaphores & Monitors

Today

- Semaphores
- Monitors
- … and some other primitives

Next time

- Deadlocks

Last time - locks

- Memory objects with two operations
  - `acquire()` – Prevents progress of the thread until the lock can be acquired
  - `release()`
- Come in two varieties – spinning and blocking
  - Spin if you expect a short wait and there are multiple cores

```
acquire (lock) {
        while(lock->held); // caller busy waits
    lock->held = 1;
}
```

  - Block if there's only one core or you expect a long wait
- Of course, both operations must be atomic
  - Need hw help – TSL or xchg

```
while(xchg(&lk->locked, 1) != 0)
```

Semaphores

- A synchronization primitive
- Higher level of abstraction than locks
- Invented by Dijkstra in ‘68 as part of THE operating system
- Atomically manipulated by two operations
  - Down(sem) /wait/P
    - Block until semaphore sem > 0, then substract 1 from sem and proceed
    - P – not really for *proberen* or *passeer* but for a made-up word *prolaag* – *“try to reduce”*
  - Up(sem) /signal/V
    - Add 1 to sem
    - V – *verhogen* – increase in Dutch

Blocking in semaphores

- Each semaphore has an associated queue of processes/threads
  - P/wait/down(sem)
    - If sem was “available” (>0), decrement sem & let thread continue
    - If sem was “unavailable” (<=0), place thread on associated queue; run some other thread

```
down(S):
   --Sem.value;
   if (Sem.value < 0){
      add this thread to Sem.L;
      block;
```

```
typedef struct {
   int value;
   struct thread *L;
} semaphore;
```

Semaphores

- **…**
  - V/signal/up(sem)
    - If thread(s) are waiting on the queue, unblock one
    - If no threads are waiting, increment sem
      - The signal is “remembered” for next time up(sem) is called
    - Might as well let the “up-ing” thread continue execution

```
up(S):
   Sem.value++;
   if (Sem.value <= 0) {
      remove a process P from Sem.L;
      wakeup(P);
   }
```

```
typedef struct {
   int value;
   struct thread *L;
} semaphore;
```

- With multiple CPUs – lock semaphore with TSL
- *But then how’s this different from previous busy-waiting?*

Semaphores

Operation	Value	Sem.L	CR
	1	{}	<>
P1 down	0	{}	P1
P2 down	-1	{P2}	P1
P3 down	-2	{P2,P3}	P1
P1 up	-1	{P3}	P2

```
down(Sem):
--Sem.value;
if (Sem.value < 0){
   add this thread to Sem.L;
   block;
}

up(Sem):
Sem.value++;
if (S.value <= 0) {
    remove a thread P from Sem.L;
    wakeup(P);
}
```

Types of semaphores

- Binary semaphores – mutex
  - Sem is initialized to 1
  - Used to guarantee mutual exclusion
  - Useful with thread packages

```
mutex_lock:
    TSL REGISTER, MUTEX
    CMP REGISTER, #0
    JXE ok
    CALL thread_yield
    JMP mutex_lock
ok: RET
```

```
mutex_unlock:
    MOVE MUTEX, #0
    RET
```

- Counting semaphores
  - Let *N* threads into critical section, not just one
  - Sem is initialized to *N*, number of (identical) units available
  - Allow threads to enter as long as there are units available

Semaphores

- Using both counting semaphores and mutexs

```
semaphore empty, // # of empty buffers, set to all
          full,  // count of full buffers, set to 0
          mutex; // initially 1
```

Producer

```
while (TRUE){
    item = produce_item();
    down(empty);
    down(mutex);
    insert_item(item);
    up(mutex);
    up(full);
}
```

Consumer

```
while (TRUE){
    down(full);
    down(mutex);
    item = remove_item();
    up(mutex);
    up(empty);
    consume_item(item);
}
```

Readers-writers problem

- Model access to database
- One shared database
  - Multiple readers allowed at once
  - Only one writer allowed at a time
    - If writers is in, nobody else is

```
semaphore db,       // mutex for writers (only one) and
                    // first/last reader
          mutex;    // mutual exclusion for rc upate
int rc;             // read count or number of readers in

void writer(void)
{
   while(TRUE) {
     think_up_data();
     down(&db);
     write_db();
     up(&db);
   }
}
```

Readers-writers problem

```
void reader(void)
{
   while(TRUE) {
      down(&mutex);
      ++rc;
      if (rc == 1) down(&db);
      up(&mutex);

      read_db();

      down(&mutex);
      --rc;
      if (rc == 0) up(&db);
      up(&mutex);

      use_data();
   }
}
```

What problem do you see for the writer?

Idea for an alternative solution: When a reader arrives, if there's a writer waiting, the reader could be suspended behind the writer instead of being immediately admitted.

Mutexes in Pthreads

- Basic mechanism – mutex

```
pthread_mutex_init – create it
pthread_mutex_destroy – destroy it
pthread_mutex_lock – acquire it or block
pthread_mutex_trylock – acquire or fail (you can spin then)
pthread_mutex_unlock – release it
```

- Also supports conditions variables
  - Typically used to block threads until a condition is met
  - Must always be associated with a mutex to avoid a race condition between a thread preparing to wait and another one signaling it (signal arriving before the thread is actually waiting)

```
pthread_cond_init – create it
pthread_cond_destroy – destroy it
pthrad_cond_wait – yield until the condition is satisfied
pthread_cond_signal – restart one of the threads waiting on it
pthread_broadcast – restart all threads waiting on it
```

Mutexes in Pthreads

```
pthread_mutex_t mutex;
pthread_cond_t condc, condp;

void *producer(void *ptr)
{
   int i;

   for (i = 1; i <= MAX; i++) {
     pthread_mutex_lock(&mutex);
     while (buffer !=0) pthread_cond_wait(&condp, &mutex);
     buffer = i;
     ptread_cond_signal(&condc);    /* wakeup consumer */
     pthread_mutex_unlock(&mutex);
   }
   pthread_exit(0);
}

void *consumer(void *ptr)
{
   int i;

   for (i = 1; i <= MAX; i++) {
     pthread_mutex_lock(&mutex);
     while (buffer ==0) pthread_cond_wait(&condc, &mutex);
     buffer = 0;
     ptread_cond_signal(&condp);   /* wakeup producer */
     pthread_mutex_unlock(&mutex);
   }
   pthread_exit(0);
}
```

Clearly missing a few definitions, including `main`

Problems with semaphores & mutex

- Solves most synchronization problems, but:
  - Semaphores are essentially shared global variables
    - Can be accessed from anywhere (bad software engineering)
  - No connection bet/ the semaphore & the data controlled by it
  - Used for both critical sections & for coordination (scheduling)
  - No control over their use, no guarantee of proper usage

"Minor" change?

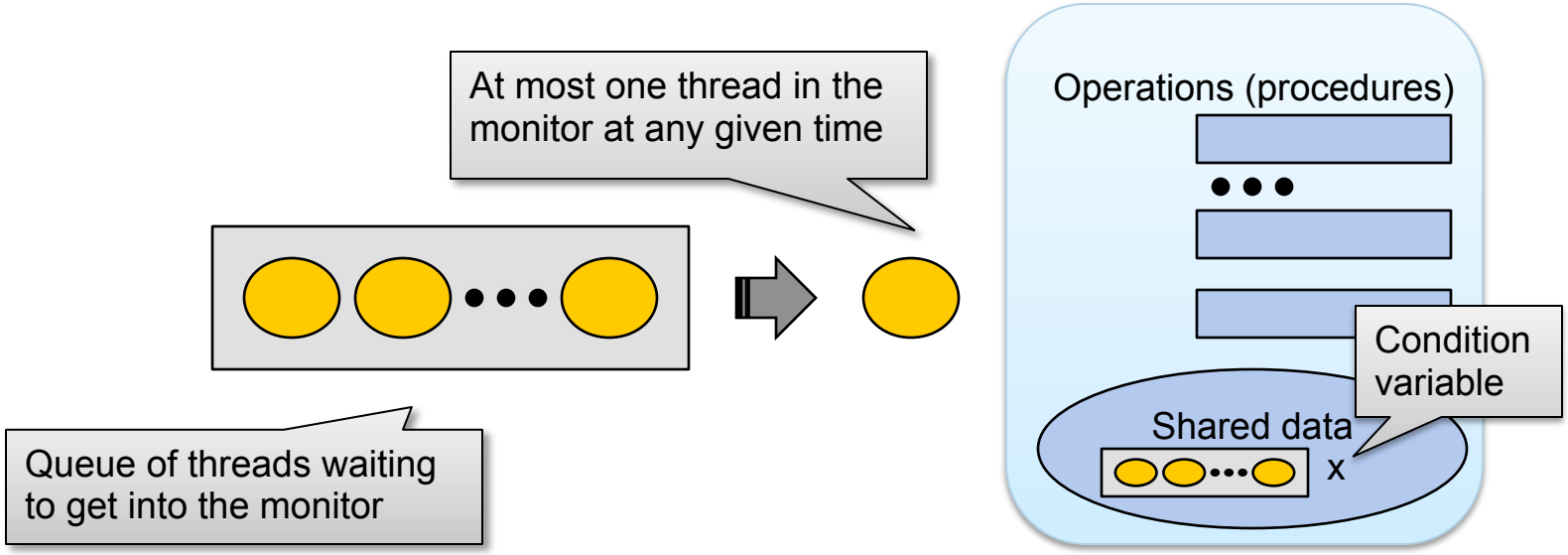
```
// producer
while (TRUE){
    item = produce_item();
    down(empty);
    down(mutex);
    insert_item(item);
    up(mutex);
    up(full);
}
```

```
// producer
while (TRUE){
    item = produce_item();
    down(mutex);
    down(empty);
    insert_item(item);
    up(mutex);
    up(full);
}
```

What happens if the buffer is full?

Monitors

- Monitors - higher level synchronization primitive
  - A programming language construct
    - Collection of procedures, variables and data structures
  - Monitor's internal data structures are private
- Monitors and mutual exclusion
  - Only one process active at a time - *how?*
  - Synchronization code is added by the compiler



Monitors

- Once inside a monitor, a thread may discover it can't continue, and
  - wants to wait, or
  - inform another one that some condition has been satisfied
- To enforce sequences of events – Condition variables
  - Can only be accessed from within the monitor
  - Two operations – `wait & signal`
  - A thread that waits "steps outside" the monitor (to a wait queue associated with that condition variable)
  - What happen after the signal?
    - Hoare – process awakened run, the other one is suspended
    - Brinch Hansen – process doing the signal must exit the monitor
    - *Third option? Process doing the signal continues to run (Mesa)*
  - `Wait` is not a counter – signal may get lost

Monitors in Java

- Not truly a monitor
  - Every object contains a lock
  - The synchronized keyword locks that lock
  - Can be applied to methods or blocks of statements
- Synchronized method
  – e.g. atomic integer

```
public class atomicInt {
    int value;
    …
    public synchronized postIncrement() {
        return value++;
    }
…
```

- Synchronized statements
  - You can lock any object, and have the lock released when you leave the block of statements

```
void foo (ArrayList list) {
  …
  synchronized(list) {
    // manipulate list now
  }
…
}
```

Message passing

- IPC in distributed systems
- Message passing

```
send(dest, &msg)
recv(src, &msg)
```

- Design issues
  - Lost messages: acks
  - Duplicates: sequence #s
  - Naming processes
  - Performance
  - …

Producer-consumer with message passing

```
#define N 100    /* num. of slots in buffer */

void producer(void)
{
   int item; message m;

   while(TRUE) {
      item = produce_item();
      receive(consumer, &m);
      build_message(&m, item);
      send(consumer, &m);
    }
}
```

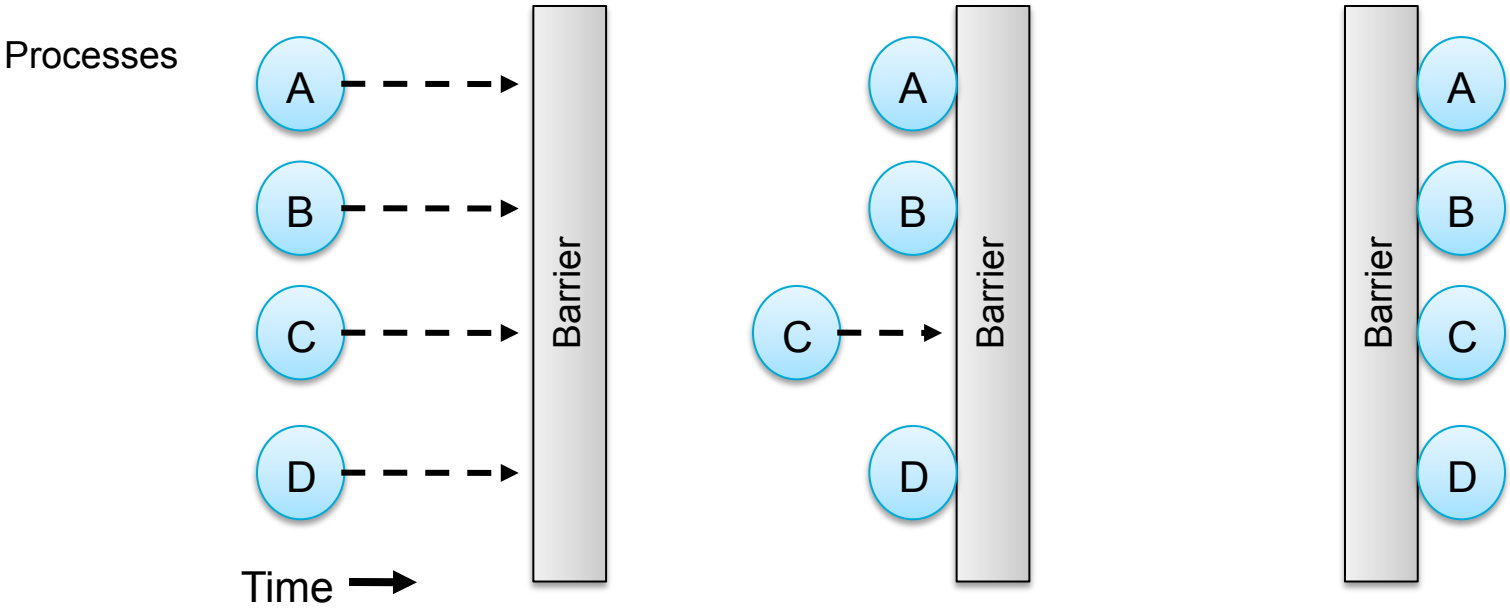
```
void consumer(void)
{
   int item, i; message m;

   for(i = 0; i < N; i++)
         send(producer, &m);

   while(TRUE) {
       receive(producer, &m);
       item = extract_item(&m);
       send(producer, &m);
       consume_item(item);
     }
}
```

Barriers

- To synchronize groups of processes
- Type of applications
  - Execution divided in phases
  - Process cannot go into new phase until all can



- e.g. Temperature propagation in a material

Join

- Sometimes you want to wait until a thread has terminated – *join()*
- A common use:
  - Start *N* threads
  - Sit in a loop waiting for all threads …
    - It really doesn't matter much which one finishes first, you just wait in an arbitrary order
- Similar but not the same as a barrier
  - join() waits until threads have terminated, and so given up all their resources
  - A barrier is achieved before threads have terminated

Dining philosophers problem

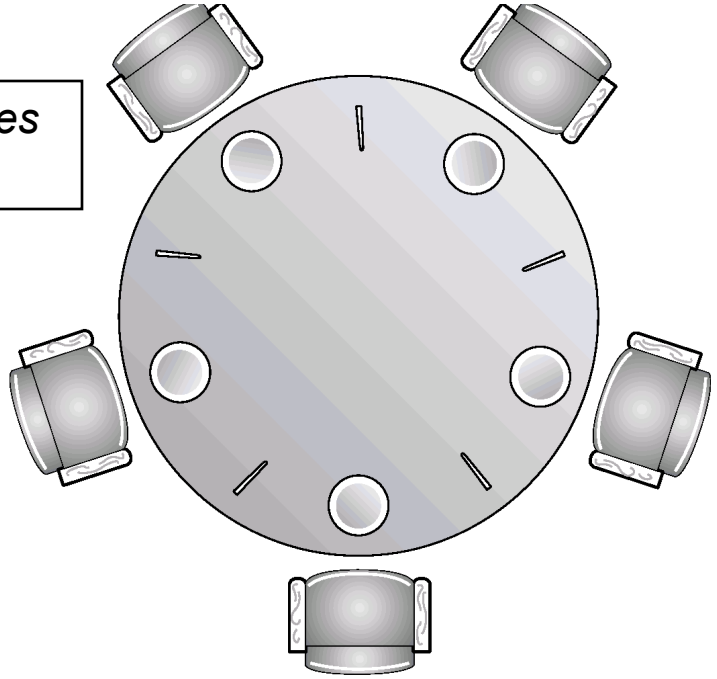
- Philosophers eat/think
- To eat, a philosopher needs 2 chopsticks
- Picks one at a time
- *How to prevent deadlock*

```
#define N 5

void philosopher(int i)
{
   while (TRUE) {
      think();
      take_chopstick(i);
      take_chopstick((i+1)%N);
      eat();
      put_chopstick(i);
      put_chopstick((i+1)%N);
   }
}
```

Now: Everybody takes the left chopstick!

Why not just protect all this with a mutex?



Nonsolution

Dining philosophers example

```
void philosopher(int i)
{
  while(TRUE) {
    think();
    take_chopstick(i);
    eat();
    put_chopstick(i);
  }
}

void take_chopstick(int i)
{
   down(&mutex);
   state[i] = HUNGRY;
   test(i);
   up(&mutex);
   down(&s[i]);
}
```

```
void put_chopstick(int i)
{
  down(&mutex);
  state[i] = THINKING;
  test(LEFT);
  test(RIGHT);
  up(&mutex);
}

void test(int i)
{
  if ((state[i] == hungry &&
     state[LEFT] != eating &&
     state[RIGHT] != eating) {
     state[i] = EATING;
     up(&s[i]);
  }
}
```

state[] – too keep track of philosopher's
state (eating, thinking, hungry)
s[] – array of semaphores, one per philosopher

Dining philosophers with monitors

```
void philosopher(int i)
{
  while(TRUE) {
    dp.take_chopstick(i);
    eat();
    dp.put_chopstick(i);
  }
}

Monitor dp
{
  enum {EATING, HUNGRY, EATING}
   state[5];
  condition s[5];

  void take_chopstick(int i)
  {
     state[i] = HUNGRY;
     test(i);
     if (state[i] != EATING)
       s[i].wait();
  }
```

```
  void put_chopstick(int i)
  {
    state[i] = THINKING;
    test(LEFT); test(RIGHT);
  }

  void test(int i)
  {
    if ((state[i] == HUNGRY &&
       state[LEFT] != EATING &&
       state[RIGHT] != EATING) {
       state[i] = EATING;
       s[i].signal();
    }
  }

  void setup()
  {
    for (i = 0; i < 5; i++)
      state[i] = THINKING;
  }
} /* end Monitor dp */
```

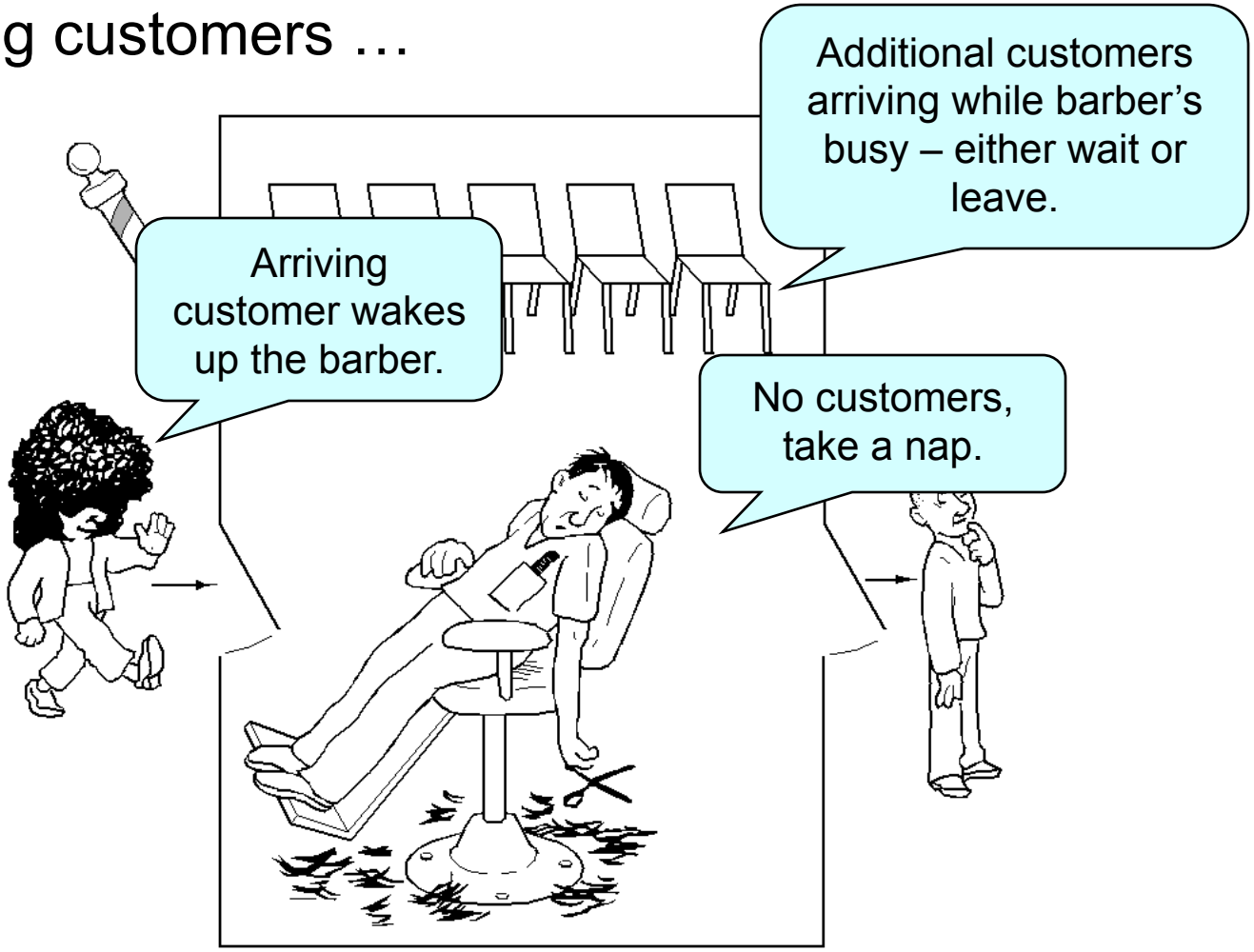
Coming up

- Deadlocks

  How deadlock arise and what you can do about them

The sleeping barber problem

One barber, one barber chair and *n* chairs for waiting customers …



The sleeping barber problem

```
#define CHAIRS 5

void barber (void)
{
   while (TRUE) {
      …
      …
      /* sleep if no customers */
      --waiting;
      …
      …
      cut_hair();
   }
}
```

```
void customer (void)
{
   …
   if (waiting < CHAIRS) {
        ++waiting; /* sit down */
        …
        …
        …
        get_haircut();
   } else { /* go elsewhere */
        …
   }
}
```

Semaphores:

- **Customer – count waiting customers (excluding the one in the barber chair)**
- **Barbers – number of barbers who are idle**
- **mutex – for mutual exclusion**

The sleeping barber problem

```
#define CHAIRS 5

void barber (void)
{
   while (TRUE) {
     down(&customers);
     /* sleep if no customers */
     down(&mutex);
     --waiting;
     up(&barbers);
     up(&mutex);
     cut_hair();
   }
}
```

```
void customer (void)
{
   …
   if (waiting < CHAIRS) {
        ++waiting; /* sit down */
        …
        …
        …
        get_haircut();
   } else { /* go elsewhere */
        …
   }
}
```

Semaphores:

- **Customer – count waiting customers (excluding the one in the barber chair)**
- **Barbers – number of barbers who are idle**
- **mutex – for mutual exclusion**

The sleeping barber problem

```
#define CHAIRS 5

void barber (void)
{
   while (TRUE) {
     down(&customers);
     /* sleep if no customers */
     down(&mutex);
     --waiting;
     up(&barbers);
     up(&mutex);
     cut_hair();
   }
}
```

```
void customer (void)
{
   down(&mutex);
   if (waiting < CHAIRS) {
        ++waiting; /* sit down */
        up(&customers);
        up(&mutex);
        down(&barbers);
        get_haircut();
   } else { /* go elsewhere */
        up(&mutex);
   }
}
```

Semaphores:

- **Customer - count waiting customers (excluding the one in the barber chair)**
- **Barbers – number of barbers who are idle**
- **mutex – for mutual exclusion**