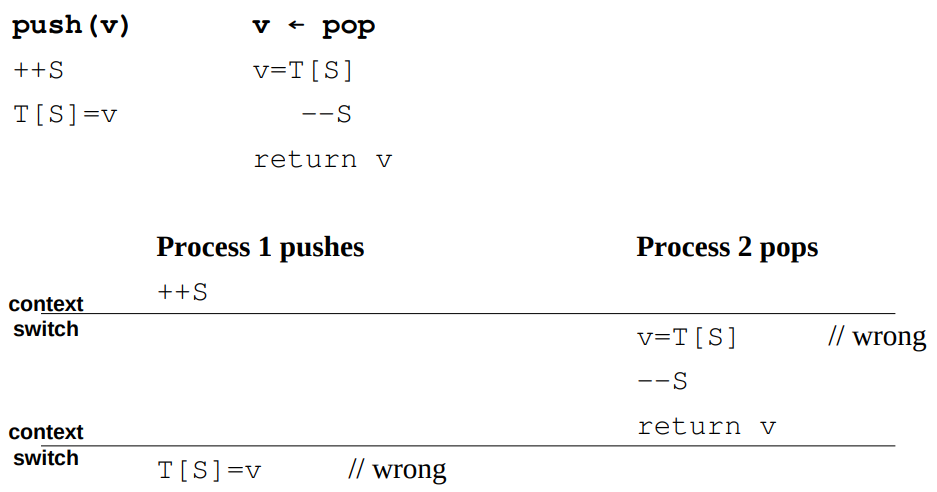
Synchronizing with semaphores: classic problems

* Producer/consumer
* Dining philosophers
* Cigarette smokers
* Synchronization barrier
* Readers/writers
* Sleeping barber

Synchronization

Example: a stack consisting of a shared array T and an index S



There is no way of stopping process scheduling or predicting it reliably.

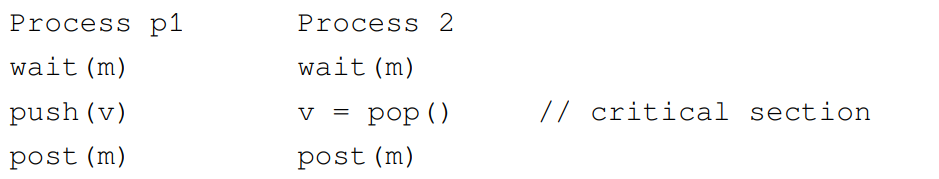
A **critical section** is a block of code using a shared resource, such as the stack in our example.

**The golden rule is that at any given moment, there must be at most one process (or thread) inside a critical section.**

This way even if the kernel schemes against us, no harm can come to our resources.

AVOID DATA CORRUPTION AND STARVATION.

Assuming that m is a semaphore initialized to 1.



wait is atomic. m is decremented by first wait.

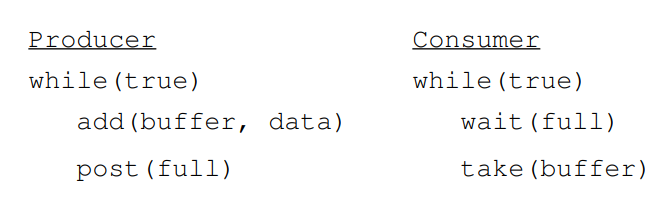
A semaphore acquiring only the values 0 and 1 is called a binary semaphore, or a mutex (mutual exclusion – only one or a number of executions be able to change the database).

Generally semaphores are used to model “a number of available resources” (that’s why they are called counting semaphores).

The **producer-consumer** model is by far the most widely encountered synchronization model. A producer process produces data, and a consumer process consumes the said data.

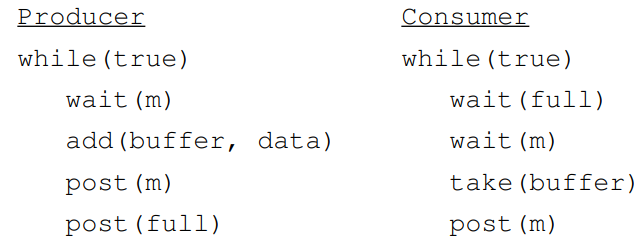
**1) Unbounded buffer case**: the consumer process must execute only if there is something to consume in the buffer; otherwise it must wait. Producer will ensure that customer will wait for the data until it is ready.

The full=0 semaphore represents the number of products in the buffer



That’s fine; but what happens if both processes enter the buffer at the same time? If there is no while loop, this won’t be a problem.

**Then the buffer becomes corrupted like our stack! We need to protect the access to the critical section through a mutex m = 1!**



With full, consumer waits for data to buffer to be available.

With m, we are checking if somebody is accessing the data.

We solved the underflow problem, but what about the overflow problem? Producer produces more than the consumer can consume.

**2) Bounded buffer case**: Now we also have an upper limit to our buffer. The producer must not produce if the buffer is full! Empty spaces are now a resource too!

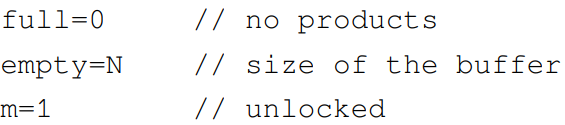
Semaphore full: number of products in the buffer – to check if buffer has data

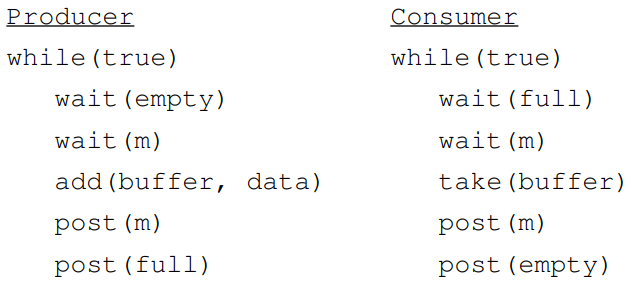
Semaphore empty: number of empty spaces in the buffer – to check if there is space in buffer

Semaphore m: concurrent access lock – to ensure data is accessed only by 1 execution, for critical region

If there is no empty space for next portion to be written in buffer, producer should be blocked until some of the data in buffer is consumed.

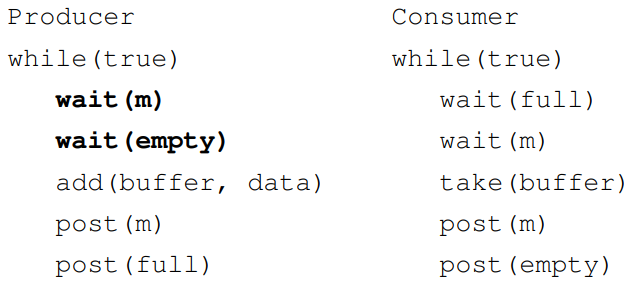
Initially:





At any given moment empty+full <= N

The order of waits is crucial! Let’s see what happens if we exchange them.

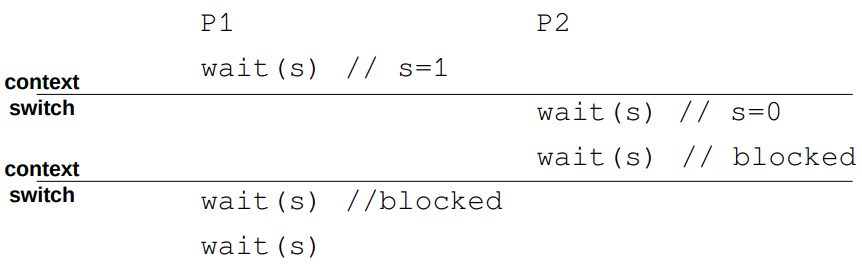


Imagine the producer getting the lock and then encountering a full buffer..the system will be blocked indefinitely!

What happens when the consumer needs more than 1 resource?

Process P1 needs 3 resources and process P2 needs 2 resources.

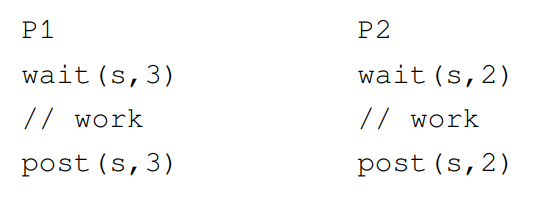
Initially we have s=2 resources.

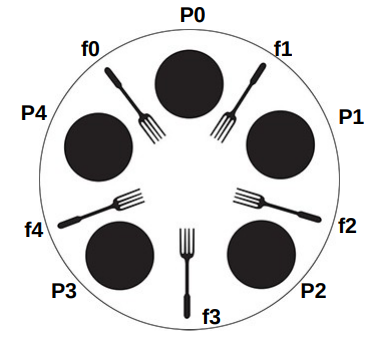


It’s a pity, process 2 could have been served with 2 resources; now they’ll have to wait until some other process calls post.

**Calling wait k times is not the same as an atomic wait decreasing the semaphore by k.**

This functionality is provided readily by System V semaphores;   
(POSIX semaphores can do it too, albeit indirectly :)



The **dining philosophers** is a classic synchronization problem introduced by Dijkstra.

Five philosophers are sitting around a dinner table, with a fork in between each pair of adjacent philosophers.

In order to eat, a philosopher needs to pick up the two forks that lie at the philosopher’s left and right sides

Each philosopher alternates between thinking (non-critical section) and eating (critical section).

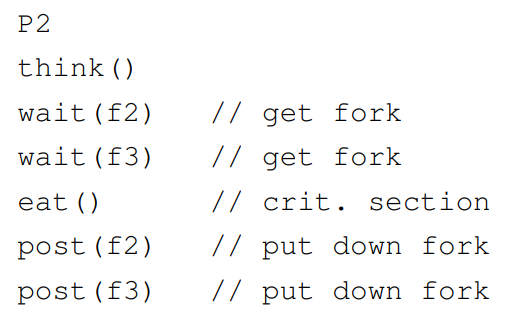
Since the forks are shared, there is a synchronization problem between philosophers (processes or threads).

The forks are our shared resources, so we’ll have a semaphore representing each of them.

We have more than 1 process (or more than 1 philosophers) at a time.

Thinking is not critical. Problem is eating, they have to use 2 forks.

A first attempt at solving the problem:



This scheduling can happen if the kernel dislikes you:

Graphical user interface, text, application

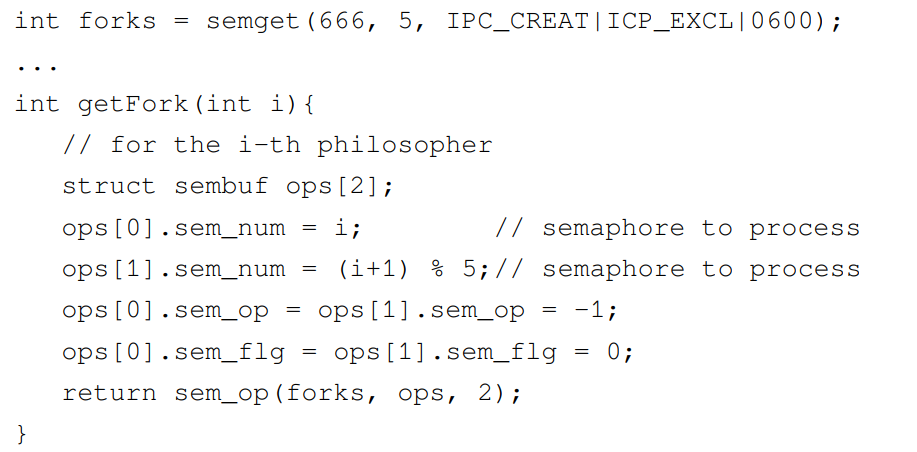
Description automatically generated

All philosophers end up starving even though 2 of them could have been served.

Main challenge: a philosopher must either get both forks, if available, or otherwise none!

We should combine both waits and both posts in 1 atomic instructions (1 for waits, 1 for posts).

Easy to solve with IPC/System V semaphores



WE INTRODUCE FORKS AS 5 SEMAPHORES.

RETURN IS 2 OF THE 5 FORKS WILL BE BLOCKED.

ops[0] and [1] are 2 forks philosopher i uses.

Another classic problem are the **cigarette smokers** (1971).

Assume a cigarette requires three ingredients to make and smoke: tobacco, paper, and matches (3 RESOURCES).

There are 3 smokers (PROCESSES) around a table, each of whom has an infinite supply of one of the 3 ingredients — one smoker has an infinite supply of tobacco, another has paper, and the third has matches.

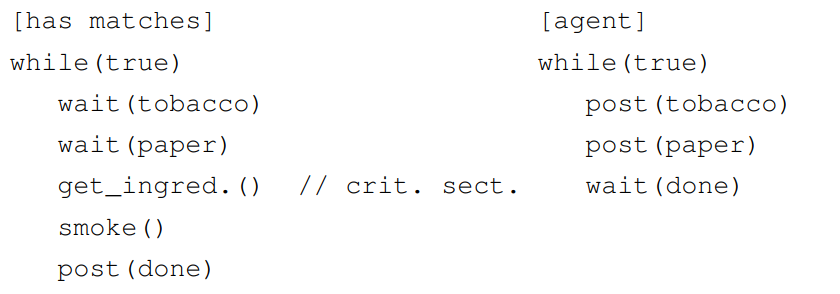
There is also a non-smoking agent who enables the smokers to make their cigarettes by arbitrarily (non-deterministically) selecting two of the supplies to place on the table.

The smoker who has the third supply should remove the two items from the table, using them (along with their own supply) to make a cigarette, which they smoke for a while AND BLOCKS OTHER 2 FROM SMOKING.

Once the smoker has finished his cigarette, the agent places two new random items on the table. This process continues forever.

The ingredients are resources so we’ll have one semaphore for each.

A first attempt



Looks good? No..

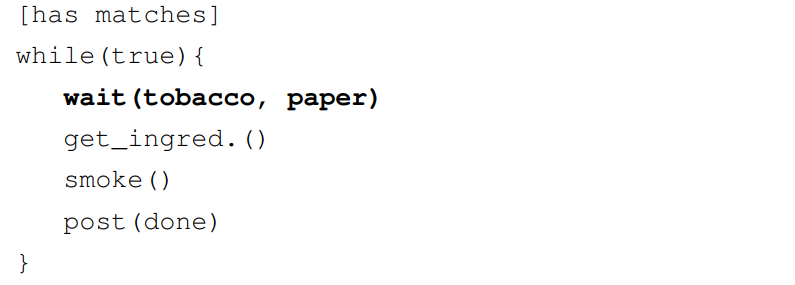
3 EXECUTIONS, ALL ARE WAITING FOR OTHERS TO RELEASE THE RESOURCE.

A picture containing graphical user interface

Description automatically generated

What if the agent brings tobacco and paper, but one smoker gets the tobacco and the other the paper? None will be able to smoke, the system will be deadlocked (good for the smokers, but for the system)!

Similarly to the dining philosophers, each smoker must either get both ingredients, if available, or otherwise none; in order to avoid effectively the deadlocks. e.g.:



AT THE END, PROCESS RELEASES ALL THE RESOURCES INCLUDING THE ONE HE/SHE HAS.

The **synchronization barrier** is another often encountered problem. We have N processes (or threads), and any of them reaching this point must stop and cannot proceed unless all other threads/processes have reached this barrier. YOU DON’T WANT ANY PROCESS TO STAY BEHIND AT A POINT.

PTHREAD LIBRARYSİNDE BU BARRIERİN SPECIFIC IMPLEMENTASYONU VAR.

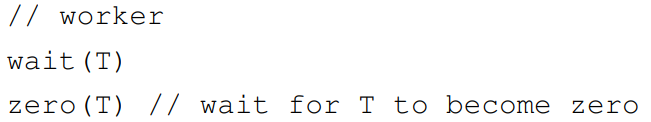
e.g. worker processes and a boss process: the boss process does not pay their salary, unless all workers have completed a required task.

Text

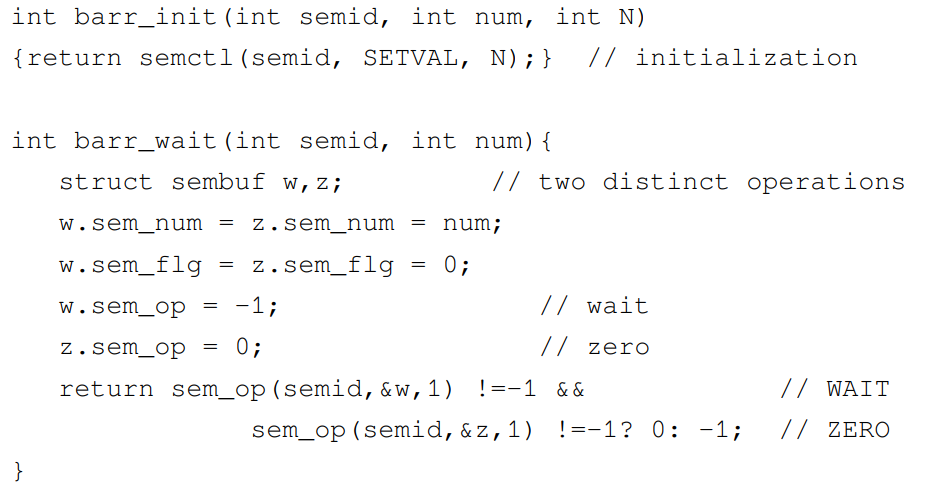
Description automatically generated with medium confidence

We have a single semaphore T initialized at N.

Every process/thread reaching the rendezvous point will call wait on it (DECREMENT BY 1), and then wait for it to become zero. If T becomes zero, that means everyone has reached the barrier.



zero is a call specific to System V/IPC semaphores.



ALL N EXECUTIONS START WAITING ARE CALLING BARR\_WAIT AT THE SAME TIME. SEMAPHORE SET WILL RELEASE ALL OF THEM. REST OF THE CODE CAN BE EXECUTED BY ALL OF THEM FROM THAT POINT ON.

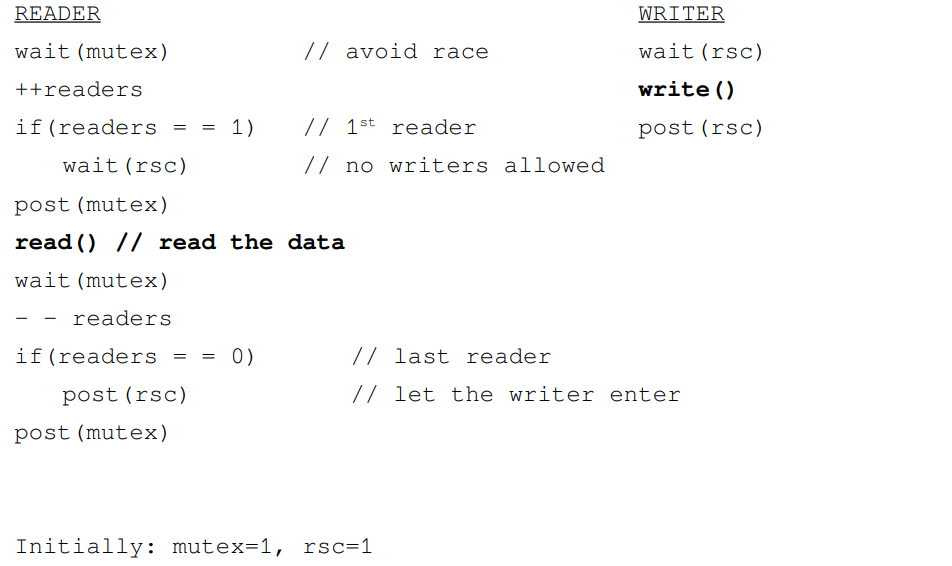
The **readers-writers** is another classic synchronization problem (1971).

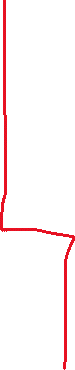
We have a shared resource that two types of processes (or threads) access:

* The readers: that do not modify the resource
* The writers: that modify the resource

Readers can access the data in any order and number they like. However

* at most one writer is allowed to write at any given moment
* of course no reader should be reading while a writer is writing.





readers: the number of active readers

rsc: makes sure we have only one writer

mutex: makes sure the shared variable readers is modified safely, to avoid other readers

MUTEX IS THERE TO ENSURE THAT IF THERE IS AT LEAST 1 READER, WRITER SHOULDN’T ACCESS TO DATA.

WHEN ALL THE READERS ARE DONE, WRITER SHOULD ACCESS THE DATA.

READERS HAVE PRIVILEGE.

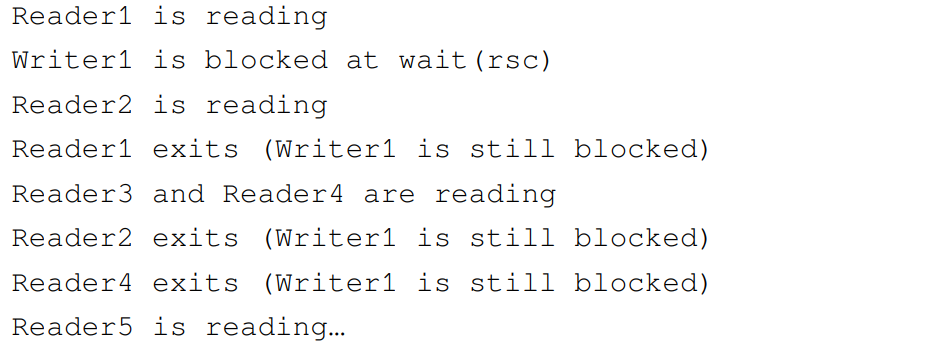
WHEN YOU READ(), YOU DON’T CORRUPT ANYTHING.

While a writer is writing, the first reader will be blocked at wait(rsc) and the subsequent ones at wait(mutex)

In the database world this is known as a “lock”.

Readers ask the Database Managament System (DBMS) for a “shared lock” and writers ask for a “exclusive lock”.

However, imagine the following scenario:



i.e. if the readers are too many, a writer might have to wait indefinitely.

Solution: prioritize writers!

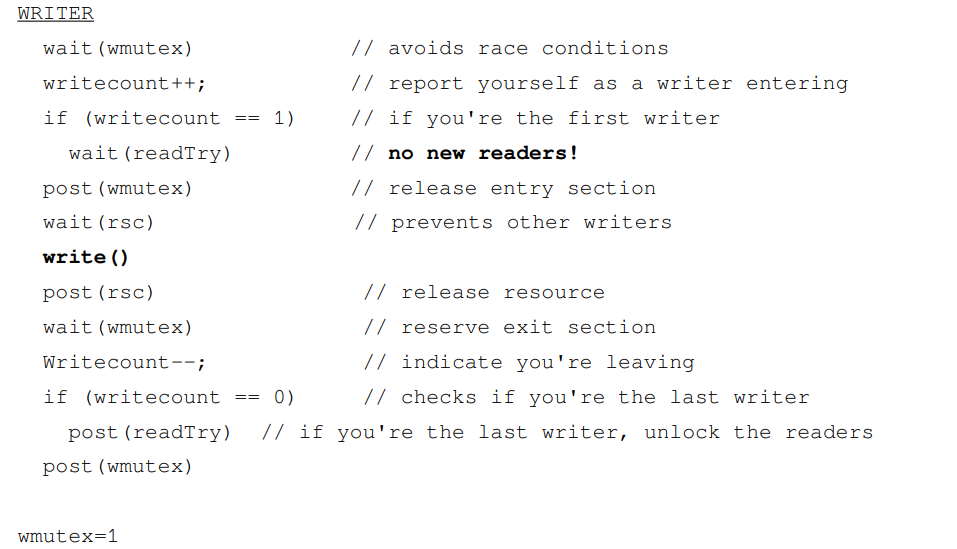
i.e. no writer, once added to the queue, shall be kept waiting longer than absolutely necessary. This is also called writers-preference.

This is accomplished by forcing every reader to lock and release a “readtry” semaphore individually. The writers on the other hand don't need to lock it individually.

Only the first writer will lock the “readtry” and then all subsequent writers can simply use the resource as it gets freed by the previous writer. The very last writer must release the “readtry” semaphore, thus opening the gate for readers to try reading.

Text

Description automatically generated



WHEN ALL THE WRITERS ARE DONE, WE ENABLE READERS TO READ. WRITERS HAVE PRIVILEGE.

PTHREAD LIBRARY HAS READER PRIVILEGED SYNCHRONIZATION READ/WRITE LOCKS AND WRITER PRIVILEGED SYNCHRONIZATION READ/WRITE LOCKS MECHANISM SPECIALLY TO AVIOD THESE KIND OF SITUATIONS.

Diagram

Description automatically generatedAnother famous problem is the **sleeping barber**; also attributed to Dijkstra (1965).

The barber shop has one barber and two rooms. The waiting room with N chairs, and the cutting room with a single chair.

**Barber**: When he finishes cutting a customer's hair, he dismisses the customer and goes to the waiting room to see if there are others waiting. If there are, he brings one of them back to the chair and cuts his hair. If there are none, he returns to the chair and sleeps in it.

**Customer**: each customer, when he arrives, looks to see what the barber is doing. If the barber is sleeping, the customer wakes him up and sits in the cutting room chair. If the barber is cutting hair, the customer stays in the waiting room. If there is a free chair in the waiting room, the customer sits in it and waits his turn. If there is no free chair, the customer leaves.

All actions (cutting hair, etc) can take an unknown amount of time. This can cause a lot of issues.

BERBER TRAŞ EDERKEN MÜŞTERİ GELDİ WAITING ROOMA OTURACAK. OTURMADAN BERBER İŞİ BİTİRDİ. BERBER DAHA KİMSE OTURMADIĞI İÇİN GİDİP UYUDU. BEKLEYEN DE BERBER BİRİNİ TRAŞ EDİYOR DİYE (NE DE OLSA BAKTI), GİTTİ WAITING ROOMA OTURDU. SONRA GELENLER BİRİ BEKLEDİĞİ İÇİN DİREKT SIRAYA GİRDİLER.

**Issues**: for instance a customer may arrive and observe that the barber is cutting hair, so he goes to the waiting room. While they're on their way, the barber finishes their current haircut and goes to check the waiting room. Since there is no one there (the customer not having arrived yet), he goes back to their chair and sleeps. The barber is now waiting for a customer, but the customer is waiting for the barber.

Or, two customers may arrive at the same time when there happens to be a single seat in the waiting room. They observe that the barber is cutting hair, go to the waiting room, and both attempt to occupy the single chair.

BERBERİN TRAŞ YAPIP YAPMADIĞINI KONTROL SIRASINDA BAŞKA PROCESS GELMEMELİ.

BERBER TRAŞ YAPIYORSA İŞİ BİTİRİNCEYE KADAR GELEN MÜŞTERİ BEKLEMELİ.

İÇERİDE MÜŞTERİ OLMADIĞI İÇİN GELEN HERKES SIRAYA GİRMEMELİ.

BERBER CUSTOMER VAR MI YOK MU DİYE BAKARKEN WAIT ETMELİ.

mutex 🡪 BINARY, GENELDE BÖYLE. –‘YE GİDENLERİ DE VAR. GENELDE BU BARİYER İÇİN KULLANILIRDI.

accessWRSeats 🡪Seatlere access var mı

// **mutex**; whether the barber is ready to cut or not

Semaphore barberReady = 0

// **mutex** to control access to the number of waiting room seats

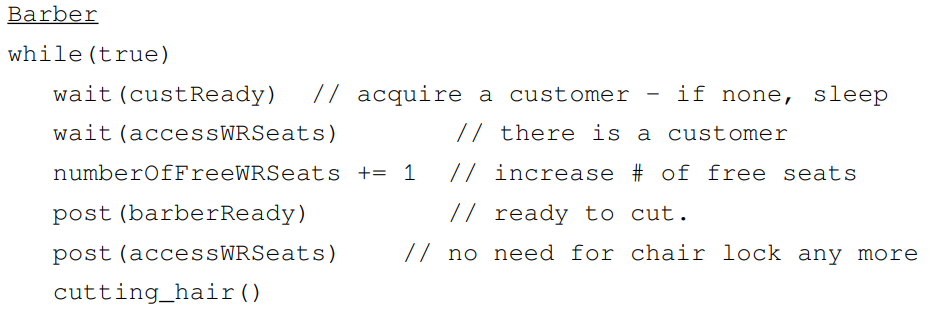
Semaphore accessWRSeats = 1 (do I have access to seat)

// the number of customers currently waiting at the waiting room

Semaphore custReady = 0

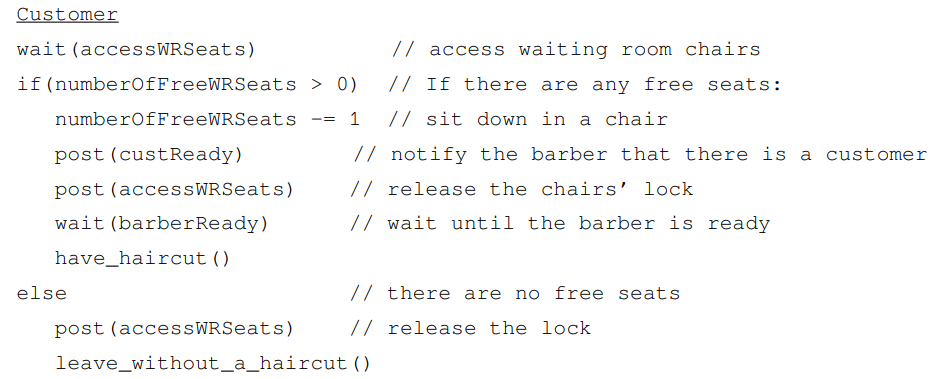
// total number of free seats in the waiting room

int numberOfFreeWRSeats = N



SIRAYLA:

* MÜŞTERİ VAR MI DİYE WAITING ROOM’A BAK, YOKSA MÜŞTERİ GELENE KADAR UYU
* MÜŞTERİ VARSA BOŞ KOLTUK SAYISINI 1 AZALT
* BERBER HAZIR DE
* KOLTUĞA ULAŞIM HAZIR DE
* TRAŞ ET



SIRAYLA:

* YER VAR MI DİYE BAK, WAITING ROOM KOLTUĞUNA ACCESS ET
* YER VARSA GİT OTUR
* BERBERE HAZIRIM DE
* YERİ BOŞALT
* BERBERİN HAZIR OLMASINI BEKLE

CUSTREADY İLE BERBERİN İÇERİ BAKTIĞINDA KİMSEYİ GÖRMEMESİ SORUNUNU ÇÖZDÜK.

BARBERREADY VE ACCESSWRSEATS İLE CUSTOMER PROBLEMİNİ ÇÖZDÜK.