Lecture 20 — Advanced Concurrency Problems

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

April 8, 2020

Get A Pizza This!

Image Credit: Valerio Capello

The Pizza Makers Problem

Let's consider a more advanced concurrency problem.

It's also called the "Cigarette Smokers Problem".

But smoking is bad for you.

Pizza, while not exactly health food, is amazing. Delicious, delicious pizza.

Iron Chef: Pizza

A new show to be hosted by some famous TV chef personality is being pitched, and you're going to write a simulation of it.

A pizza requires three ingredients: dough, sauce, and cheese.

This Could Be a Real Show

Each contestant has an unlimited supply of one ingredient.

Contestant A has an unlimited supply of dough, Contestant B has an unlimited supply of sauce, and Contestant C has an unlimited supply of cheese.

Each contestant needs to get the two ingredients they do not have and then can make a pizza.

They will continue to (try to) make pizza in a loop until time is up.

Start the Show!

At the beginning of the episode, the host places two different random ingredients out.

Contestants can signal the host to ask for more ingredients, but they should not do so unless they actually need some.

Each time the host is woken up (signalled), he again places two different random ingredients out.

When an ingredient is placed on the table, the host posts on the associated semaphore.

Is this Relevant?

In this scenario, there are resources provided by some external system and the contestants are processes that want resources.

But we shouldn't be wasteful: resources should only be requested when they are needed and processes should take only what they need.

Applications should only wake up if they can do something useful.

How Hard Do You Want It To Be?

There are some restrictions, though, and they could make the problem either impossible or trivial.

In the impossible version, you can't modify what the host does (which is sensible, because you don't control the other system).

But you also cannot use conditional (if) statements, which is pretty ridiculous.

All Too Easy

In the trivial version, the host tells which contestants whose turn it is.

It requires the host system to know too much about the contestants.

The interesting version has just the restriction that we can't control the host behaviour.

And he is a TV chef personality, after all, they do wacky things.

Let's Make Some Pizza!

All semaphores start at 0, except for `host` which starts as 1 (so the host will run the first time). Does this work?

Contestant A

```
wait( sauce )
get_sauce()
wait( cheese )
get_cheese()
make_pizza( )
post( host )
```

Contestant B

```
wait( dough )
get_dough()
wait( cheese )
get_cheese()
make_pizza( )
post( host )
```

Contestant C

```
wait( sauce )
get_sauce()
wait( dough )
get_dough()
make_pizza( )
post( host )
```

Him Again!

No. Deadlock can easily occur.

Suppose the host puts out sauce and dough.

If contestant B takes the dough and contestant A takes the sauce, then both of them are blocked and nobody can proceed and nobody gets pizza.

This Isn’t What I Ordered…

Part of the problem here is that a contestant doesn’t have a good way to assess what the ingredients are before going up there.

And once there, if it finds that the ingredients match someone else’s needs, we don’t call that contestant over.

That would be clever, but to make this work we would need a way to “check” what ingredients are there and semaphores don’t let us do that.

Can we work with what we have?

We Need a Sidekick!

Imagine now that each contestant gets a helper.

The job of the helper is to, well, help their contestant to make pizza by figuring out whose turn it is.

Boolean Variables

For this there are boolean variables `dough_present`, `sauce_present`, and `cheese_present`

They are all initialized to `false`.

They are protected by a semaphore (called `mutex`).

The helpers update that variable, and based on the information available, signal which contestant should come up to the table and take ingredients.

Working with a Sidekick

Each contestant now has a semaphore (such as `contestantA` for contestant A) which the helpers will post on.

Contestants are still responsible for telling the host to put out more ingredients.

Helper 1

```
wait( sauce )
wait( mutex )
if dough_present
  dough_present = false;
  post( contestantC )
else if cheese_present
  cheese_present = false;
  post( contestantA )
else
  sauce_present = true;
end if
post( mutex )
```

Helper 2

```
wait( dough )
wait( mutex )
if sauce_present
  sauce_present = false;
  post( contestantC )
else if cheese_present
  cheese_present = false;
  post( contestantB )
else
  dough_present = true;
end if
post( mutex )
```

Helper 3

```
wait( cheese )
wait( mutex )
if dough_present
  dough_present = false;
  post( contestantB )
else if sauce_present
  sauce_present = false;
  post( contestantA )
else
  cheese_present = true;
end if
post( mutex )
```

Analyzing the Helper

So let's analyze Helper 1.

In this case, the helper is woken up when sauce is placed on the table.

It then locks the mutex so that it can manipulate the shared variables of what ingredients are present.

Now we decide what to do here.

Obviously, each of the other helpers will signal the appropriate contestant based on its assessment of the state of the ingredients.

Working with a Sidekick

Contestant A

```
wait( contestantA )
get_sauce()
get_cheese()
make_pizza( )
post( host )
```

Contestant B

```
wait( contestantB )
get_dough()
get_cheese()
make_pizza( )
post( host )
```

Contestant C

```
wait( contestantC )
get_sauce()
get_dough()
make_pizza( )
post( host )
```

The contestant code is pretty much trivial now: wait until a helper signals, then go take your ingredients and make a pizza.

Once the pizza is in the oven, indicate that you are ready for more ingredients.

Rule Change

The generalized version of the problem is what happens when the host puts out ingredients periodically, without a need to be signalled to ask for more.

How do we modify the solution to deal with that?

If there is no longer a need for the contestants to signal that they want more resources, `post( host )` in the contestant code has to be removed.

But what about the helpers?

How Many Dough?

Instead of boolean variables to indicate the presence or absence of an ingredient what we need instead is an integer counter to know how many there are.

They are: `num_dough`, `num_sauce`, and `num_cheese`, and they all start as zero.

Updated Sidekick Code

Helper 1

```
wait( sauce )
wait( mutex )
if num_dough > 0
  num_dough--
  post( contestantC )
else if num_cheese > 0
  num_cheese--
  post( contestantA )
else
  num_sauce++
end if
post( mutex )
```

Helper 2

```
wait( dough )
wait( mutex )
if num_sauce > 0
  num_sauce--
  post( contestantC )
else if num_cheese > 0
  num_cheese--
  post( contestantB )
else
  num_dough++
end if
post( mutex )
```

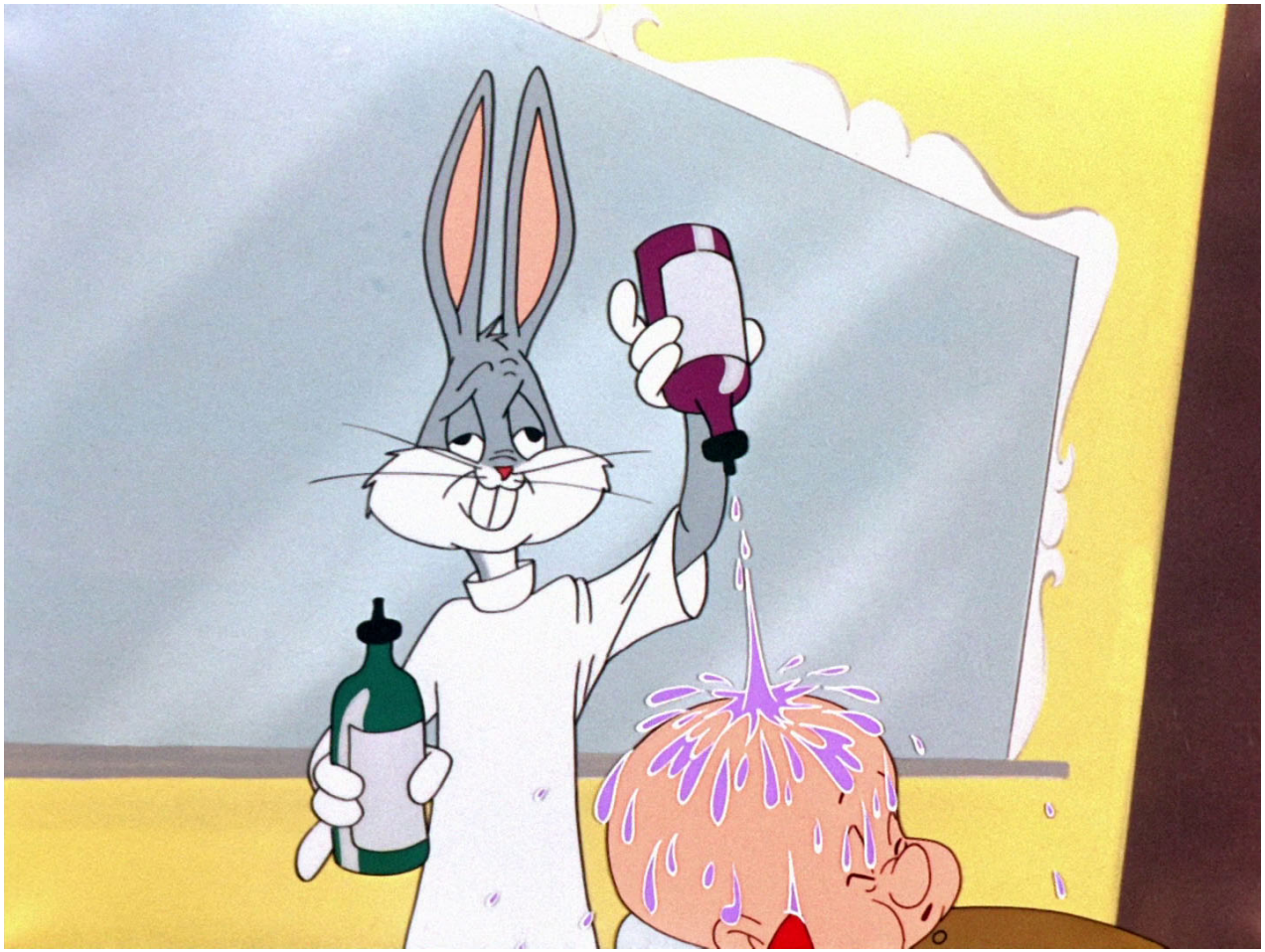
Helper 3

```
wait( cheese )
wait( mutex )
if num_dough > 0
  num_dough--
  post( contestantB )
else if num_sauce > 0
  num_sauce--
  post( contestantA )
else
  num_cheese++
end if
post( mutex )
```

This pattern is referred to as the “scoreboard”.

As threads go about their actions, they take a look at the current state (the scoreboard) and decide how to act based on that.

Barber of Seville



The Barbershop Problem

Consider the “Barbershop Problem”, originally proposed by Dijkstra.

A barbershop is a place where customers get their hair cut.

A barbershop consists of a waiting area with *n* chairs, and a barber chair.

Barbershop Behaviour

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.

Wake Up!

Customer threads should call `get_hair_cut()` when it is their turn.

If the shop is full, the customer should `return` (exit/leave).

The barber thread will call `cut_hair`.

The barber can cut only one person's hair at a time, so there should be exactly one thread calling `get_hair_cut()` concurrently.

You can assume that external forces cause customers to appear and the barber to keep working.

Scoreboard Pattern Again

We need an integer counter for customers waiting called `customers` that starts at 0.

We will also have a mutex for controlling access to `customers` called `mutex` (it obviously starts at 1).

Finally, two semaphores, `customer` and `barber` that both start at 0.

Let's write down some outline of the barber and customer code...

Barbershop Solution

Customer

```
wait( mutex )
if customers == n+1
    signal( mutex )
    return
end if
customers++
signal( mutex )

signal( customer )
wait( barber )
get_hair_cut()

wait( mutex )
customers--
signal( mutex )
```

Barber

```
wait( customer )
signal( barber )
cut_hair()
```

Is there a risk of deadlock?

Starvation?

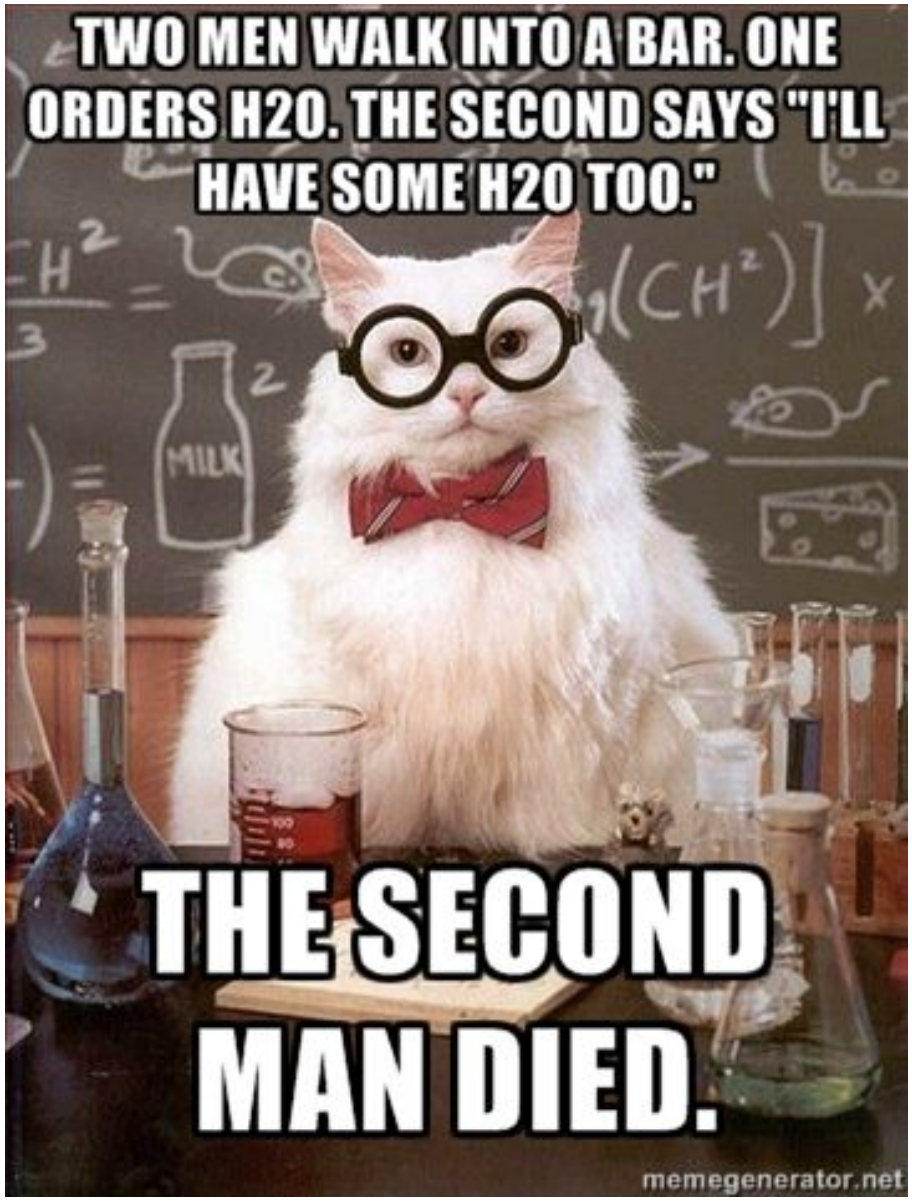
I WANT TO SPEAK TO A MANAGER

We can maybe say that customers who give up in frustration are disappointed.

Is that better than making them wait forever?

This is actually a good lesson for services in general.

SCIENCE!!!


TWO MEN WALK INTO A BAR. ONE ORDERS H2O. THE SECOND SAYS "I'LL HAVE SOME H2O TOO."
THE SECOND MAN DIED.
MILK
memegenerator.net

Science!

There are two kinds of thread, `oxygen()` and `hydrogen()`.

As you will recall from basic chemistry, water, H_2O, requires two hydrogen modules and one oxygen module.

To assemble the desired molecule (water) a group rendezvous pattern is needed to make each thread wait until all ingredients are present in the correct amounts.

As each thread passes the barrier, it should call the function `bond()` which makes the water.

Our solution must function so that all threads for one molecule invoke `bond()` before any of the threads from the next molecule do.

Water is Life

In the example, we'll assume the `oxygen` and `hydrogen` threads are created and started correctly and in the correct proportions.

The code for the creation of those two types of threads is not shown.

The reusable two-phase barrier from earlier has also been converted into C code.

The oxygen queue and hydrogen queue start as "locked".

Water is Life

```
int oxygen;
int hydrogen;
pthread_mutex_t barrier_mutex;
sem_t turnstile;
int barrier_count;
int barrier_N;
sem_t bond;
sem_t oxygen_queue;
sem_t hydrogen_queue;
```

```
void barrier_enter( ) {
  pthread_mutex_lock( &barrier_mutex );
  barrier_count++;
  if ( barrier_count == barrier_N ) {
    sem_post( &turnstile );
  }
  pthread_mutex_unlock( &barrier_mutex );
  sem_wait( &turnstile );
  sem_post( &turnstile );
}

void barrier_exit( ) {
  pthread_mutex_lock( &barrier_mutex );
  barrier_count--;
  if ( barrier_count == 0 ) {
    sem_wait( &turnstile );
  }
  pthread_mutex_unlock( &barrier_mutex );
}
```

```
void* oxygen( void* ignore ) {
  sem_wait( &bond );
  oxygen++;

  if( hydrogen >= 2 ){
    sem_post( &hydrogen_queue );
    sem_post( &hydrogen_queue );
    hydrogen -= 2;
    sem_post( &oxygen_queue );
    oxygen--;
  } else {
    sem_post( &bond );
  }

  sem_wait( &oxygen_queue );
  bond();

  barrier_enter();
  barrier_exit();

  sem_post( &bond );

  pthread_exit( NULL )
}
```

Water is Life

```
void* hydrogen( void* ignore ) {
  sem_wait( &bond );
  hydrogen++;

  if( hydrogen >= 2 && oxygen >= 1 )
    sem_post( &hydrogen_queue );
    sem_post( &hydrogen_queue );
    hydrogen -= 2;
    sem_post( &oxygen_queue );
    oxygen--;
  } else {
    sem_post( &bond );
  }

  sem_wait( &hydrogen_queue );
  bond();

  barrier_enter();
  barrier_exit();

  pthread_exit( NULL )
}
```

Analyzing the Solution

It is a little strange that the hydrogen threads don’t post on bond.

Isn’t this a problem?

Posting Bond?

It turns out no, because the oxygen threads post on it unconditionally.

When a thread arrives but the water molecule cannot be formed, whether it is oxygen or hydrogen, a post on bond takes place.

Whoever waited on bond does not matter, as long as one of the threads that went into the water molecule posts on it before leaving.

As the chemical composition of water has one oxygen, the job is assigned to this molecule.

The World is Big

These are by no means all the concurrency problems in the world.

But for now we will leave it here, before we get into really obscure problems...