ECE 252: Systems Programming and Concurrency Spring 2019

Lecture 22 — Advanced Concurrency Problems

Jeff Zarnett *2019-04-15*

Get a Pizza This!

Let's consider a more advanced concurrency problem. In [Dow08] it's called the "Cigarette Smokers Problem", but smoking is bad for you so it's going to be the "Pizza Makers Problem". Pizza, while not exactly health food, is amazing. Delicious, delicious 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. The show is about making pizza. A pizza requires three ingredients: dough, sauce, and cheese. All three ingredients are necessary to make a pizza (otherwise it does not meet the definition of a pizza).

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. 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. For example, if the host puts out cheese and sauce, then the host posts on both `cheese` and `sauce`.

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.

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. In the trivial version, the host tells which contestants whose turn it is, which is boring but also unrealistic, because again 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 [Dow08]. And he is a TV chef personality, after all, they do wacky things.

Consider the following solution. 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 )
```

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.

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. If we want to do that, we'd have to break the rule about not modifying the host's behaviour. Can we work with what we have?

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

So let's analyze Helper 1. Each of the other helpers does the same thing but for a different ingredient. 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. If the current helper is the first one to run, then all the variables will be false, so we'll end up at the else block and just set `sauce_present` to true and then let the next helper run. If `dough_present` is true, then Helper 2 has already run and we know that dough is present on the table. With both sauce and dough present, we know that the ingredients present match the needs of Contestant C. If `cheese_present` is true then Helper 3 has already run and the ingredients present match the needs of Contestant A.

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

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.

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?

Obviously if there is no longer a need for the contestants to signal that they want more resources then the `post( host )` statements in the contestant code has to be removed. But what about the helpers?

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. So let's call them `num_dough`, `num_sauce`, and `num_cheese`, and they all start as zero.

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 )
```

Now instead of setting the variables to true and false, the variables tracking the ingredients are incremented and decremented. Otherwise not much has changed. This pattern is referred to as the "scoreboard" [Dow08] – there are variables keeping track of the state of the system which are viewable from all threads. As threads go about their actions, they take a look at the current state (the scoreboard) and decide how to act based on that.

The Barbershop Problem

Consider the "Barbershop Problem", originally proposed by Dijkstra. A variant of this appears in [SGG13]. 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.

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.

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 (so you do not need to write any loops). Assume `n` is initialized to an appropriate value.

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.

As for the solution:

Customer

```
wait( mutex )
if customers == n
    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()
```

The barber code is much simpler, so let's start there. The barber waits for there to be a customer. When there is

one, the barber calls the customer forward by signalling on `barber`, and then cuts hair. Simple enough.

As for the customer, `customers` is shared data, so its access should be protected by `mutex`. If we find that the number of customers is already n – that is, the waiting area is full, the customer should leave. But not without unlocking `mutex`! If the customer forgot to do that then no more customers could enter. That's a problem. Otherwise the count of customers is increased and the customer signals the semaphore `customer`. This will wake up the barber, if the barber is sleeping, but also indicates the number of customers waiting. Then the customer must wait for the barber. When the customer gets called forward by the barber, the customer can get their hair cut. Finally, when leaving, the customer decrements the number of customers counter.

Let's look over this for a risk of deadlock. We don't see anywhere, really, the nested-waits pattern that usually indicates trouble. There isn't really any scenario where a customer can get blocked waiting for another customer or the barber, and the barber code is very simple.

As for starvation, it also won't happen. Assuming there are actually customers, the barber will have work to do. And the customers will wait for a finite period before they get their hair cut. The size of the waiting area is limited so we expect that a customer doesn't get delayed indefinitely. It is possible for them to wait quite a while, though, but they will eventually get a turn.

We can maybe say that customers who give up in frustration are disappointed but that's better than making them wait forever. If they are told they won't get service today, they can go to a different barbershop or come again some other time. That's why there's the limit on the waiting area at all. In the scenario, the barber is only human and can only cut so many people's hair in a day. The exact number will vary based on how long each hair cut takes, which depends on the person whose hair is being cut. But there is a limit of some sort: at some point the barber will go home. And only so many people can fit into the waiting room anyway: if there are too many, then nobody can move around (and this violates fire code!).

This is actually a good lesson for services in general. They have a certain capacity, and when things are overloaded it's helpful to tell clients that they are unable to take any more requests right now. Clients can then choose what to do (and it's usually either "wait a while and try again" or "go to a different service"). And you can! If the game that you want to play requires a server and all servers are busy, you can play a different game (or go outside, or who knows what!).

Chemistry: Building H_2O

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.

To clarify, if an oxygen thread arrives at the barrier when no hydrogen threads are present, it has to wait for two hydrogen threads to arrive. If a hydrogen thread arrives a the barrier when no other threads are waiting, it has to wait for an oxygen thread and another hydrogen thread. It is not necessary for the threads to know what other threads they are matched with, as long as the correct elements are present in the correct proportions.

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

The oxygen queue and hydrogen queue start as "locked" so we'll only signal the threads when they are going to proceed. Before that we have a scoreboard pattern, where threads take a look at the state of the system and decide what to do.

```
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 );
}
```

```
int main( void ) {
  oxygen = 0;
  hydrogen = 0;
  barrier_count = 0;
  barrier_N = 3;

  pthread_mutex_init( &barrier_mutex, NULL );
  sem_init( &barrier_turnstile, 0, 0 );

  sem_init( &bond, 0, 1 );
  sem_init( &oxygen_queue, 0, 0 );
  sem_init( &hydrogen_queue, 0, 0 );

  /* Creation of oxygen and hydrogen threads
   not shown for space reasons */

  pthread_mutex_destroy( &barrier_mutex );
  sem_destroy( &barrier_turnstile );
  sem_destroy( &bond );
  sem_destroy( &oxygen_queue );
  sem_destroy( &hydrogen_queue );

  pthread_exit( 0 );
}
```

```
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 )
}
```

```
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 )
}
```

Let's analyze the solution then, starting with oxygen threads. When it enters, it waits on the `bond` semaphore, which in this case is used somewhat like a mutex. If there are at least two hydrogen threads waiting, then, we can unblock two of them, signal the oxygen queue, and update the counters. If there are not two hydrogens waiting, then we just signal `bond` so that the next thread will arrive. Then the thread can get in line at the oxygen queue.

The hydrogen code is more or less the same, but the if-condition is different since it's not enough for 1 hydrogen to be present (we need at least two). If we have what we need, signal the queues to release the needed molecules, and we are ready to go. If not, get in line.

When threads are released from the oxygen queue and hydrogen queue respectively, they proceed forward to the `bond()` step, and afterwards there is a barrier enter followed by exit to wait for all of them to be finished. All we really need is for all three threads to be done with `bond()` before they can exit so the barrier enter and exit happen one after the other.

Now it may be that when a thread arrives it unblocks some thread ahead of it in line. So if the oxygen that arrives

is the 2nd oxygen and there is already one waiting ahead of it, that first one proceeds. That's okay – one molecule of water is as good as any other so we don't really care which oxygen ends up in its composition.

It is a little strange that the hydrogen threads don't post on `bond`. Isn't this a problem? It turns out no, because the oxygen threads post on it unconditionally. The reasoning on this is not too complicated. When a thread arrives but the water molecule cannot be formed, whether it is oxygen or hydrogen, a post on `bond` takes place. If, however, the thread arriving is the last one necessary for bonding to take place, then one oxygen and two hydrogen proceed. 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. If we put it in hydrogen we might post on it twice.

These are by no means all the concurrency problems in the world. There are many more in [Dow08] that could be considered. But for now we will leave it here, before we get into really obscure problems...

References

[Dow08] Allen B. Downey. *The Little Book of Semaphores (2nd Edition)*. Green Tea Press, 2008.

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.