

Griffin Hurt

Undergraduate Teaching Fellow

griffhurt@pitt.edu

<https://griffinhurt.com>

Spring 2024, Term 2244

Friday 2 PM Recitation

Apr 12th, 2024

Slides adapted from
Shinwoo Kim, Martha Dixon, and Vinicius Petrucci

Department of Computer Science
School of Computing & Information
University of Pittsburgh

Recitation 11: Threads/Synchronization

- Course News
- Threads
- Synchronization
- Quiz!
- Conditional Variables

Course News

Threads Project due Friday, April 19th at 5:59PM

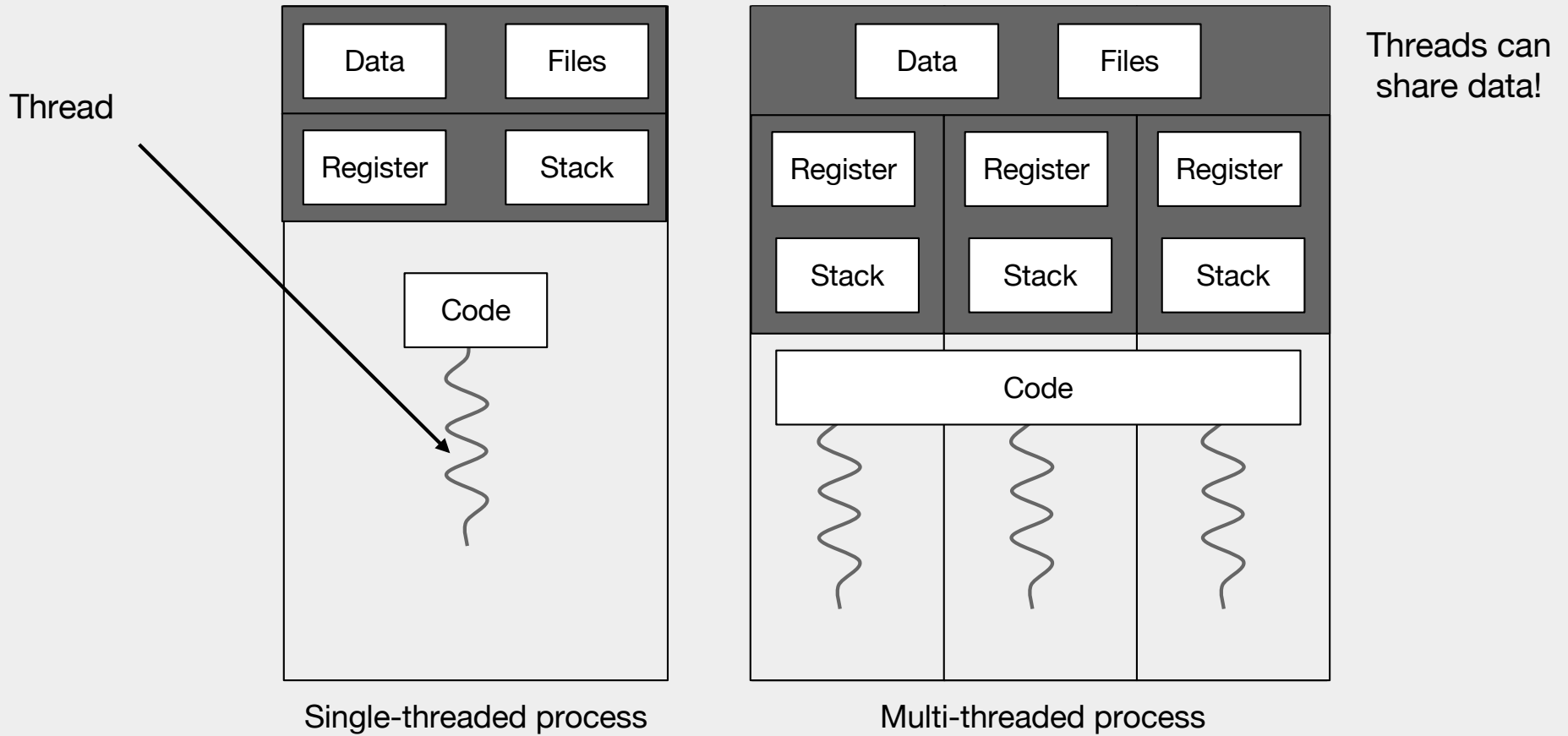
I'm going to refactor my website tonight so all slides will be posted

- Sorry for the delay!

Threads

Achieving Concurrency without `fork()`s

Processes and Threads



Posix Threads (**pthread**) – *POSIX.1c*

POSIX: Portable Operating System Interface

- Standard to unify the program and system calls that many different operating systems provide
- Provides us a 'standard library' to help create and manage threads
- `#include <pthread.h>`
- `int pthread_create(pthread_t threadID,
 FLAGS, void *(*function)(void *),
 void *restrict arg);`
- `int pthread_join(pthread_t thread, void **retval);`

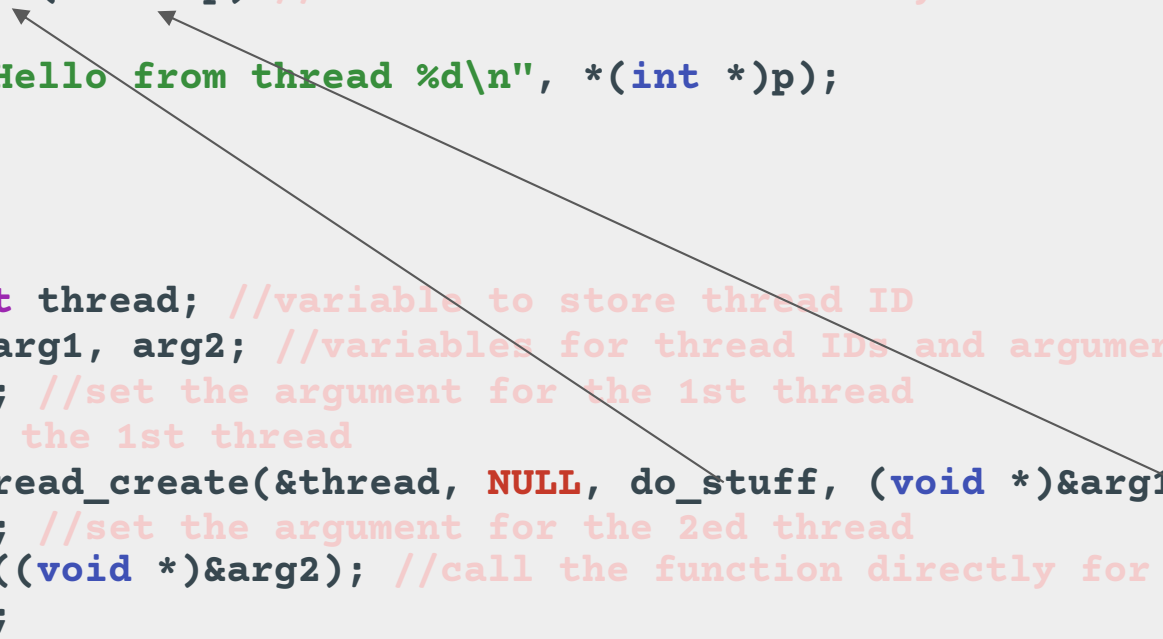
Other libraries exist

- Win32 Threads for Windows
- C11 Threads - not popular, not fully portable Thanks Microsoft
 - C++11 Threads - popular and widely used

pthread_create()

```
#include <stdio.h>
#include <pthread.h>
void *do_stuff(void *p) //function to be executed by the thread
{
    printf("Hello from thread %d\n", *(int *)p);
}

int main()
{
    pthread_t thread; //variable to store thread ID
    int id, arg1, arg2; //variables for thread IDs and arguments
    arg1 = 1; //set the argument for the 1st thread
    //create the 1st thread
    id = pthread_create(&thread, NULL, do_stuff, (void *)&arg1);
    arg2 = 2; //set the argument for the 2ed thread
    do_stuff((void *)&arg2); //call the function directly for the 2ed thread
    return 0;
}
```

A diagram consisting of two arrows. The first arrow originates from the '&arg1' argument in the pthread_create call within the main function and points to the 'p' parameter in the do_stuff function signature. The second arrow originates from the '&arg2' argument in the do_stuff call within the main function and points to the 'p' parameter in the do_stuff function signature.

Output

```
Hello from thread 2
```

pthread_create()

```
#include <stdio.h>
#include <pthread.h>
void *do_stuff(void *p)
→ {
    printf("Hello from thread %d\n", *(int *)p);
}

int main()
{
    pthread_t thread;
    int id, arg1, arg2;
    arg1 = 1;
    id = pthread_create(&thread, NULL, do_stuff, (void *)&arg1);
→ arg2 = 2;
    do_stuff((void *)&arg2);
    return 0;
}
```

When the process exits, all threads are canceled. Here, the process exited before the second thread got to print its message

pthread_yield()

```
#include <stdio.h>
#include <pthread.h>
void *do_stuff(void *p)
{
    printf("Hello from thread %d\n", *(int *)p);
}

int main()
{
    pthread_t thread;
    int id, arg1, arg2;
    arg1 = 1;
    id = pthread_create(&thread, NULL, do_stuff, (void *)&arg1);
    pthread_yield();
    arg2 = 2;
    do_stuff((void *)&arg2);
    return 0;
}
```

Output

Hello from thread 1

Hello from thread 2

`pthread_yield()` relinquishes the CPU

- Allowing another thread to assume the CPU
- Technically **deprecated**, but still portable and widely used!
 - ⇒ You shouldn't use it in your own code, but you may encounter it in the wild!

pthread_join()

```
#include <stdio.h>
#include <pthread.h>
void *do_stuff(void *p)
{
    printf("Hello from thread %d\n", *(int *)p);
}

int main()
{
    pthread_t thread;
    int id, arg1, arg2;
    arg1 = 1;
    id = pthread_create(&thread, NULL, do_stuff, (void *)&arg1);
    pthread_join(thread, NULL);
    arg2 = 2;
    do_stuff((void *)&arg2);
    return 0;
}
```

Output

Hello from thread 1

Hello from thread 2

`pthread_join(thread, NULL)` waits until `thread` terminates

Linking the `pthread` library

At compile time, need to link the POSIX thread library to your code

Using `-pthread` option to `gcc`

```
gcc -o thread_program source.c -pthread
```

- Writing a Makefile might be useful here...

Synchronization

The Dangers of Threading

Race Condition: A Refrigerator Analogy

Alice and Bob are roommates living in a dorm

- They share one refrigerator in the kitchen

Alice wakes up at 9:30 AM

- She checks the refrigerator for milk and sees that there is none
- She goes out to the store to get milk

While Alice is at the store, Bob wakes up

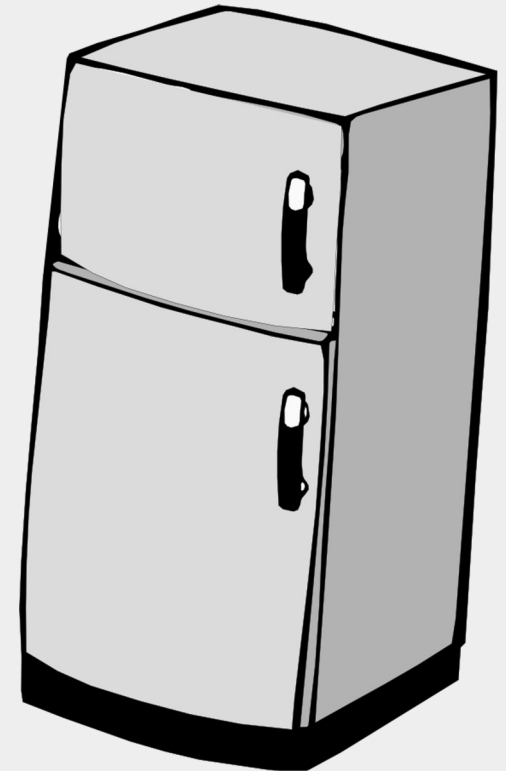
- He checks the refrigerator for milk and sees that there is none
- He goes out to the store to get milk

At 1 PM, Alice returns from the store

- And places the milk in the fridge

At 1:30 PM, Bob returns from the store

- And tries to place the milk in the fridge
- But there's already another milk in the fridge!



Synchronization

What went wrong?

Bob and Alice did not **communicate!**

Some shared resource (refrigerator)

Time delay between checking the condition (looking inside the refrigerator)

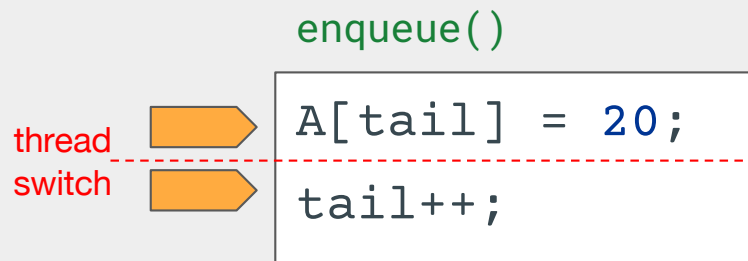
And taking an action (placing milk inside refrigerator)

Race Conditions Animated

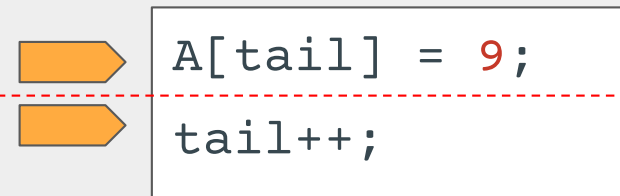
tail = 6

A[] =

1	8	5	6	9	?		
---	---	---	---	---	---	--	--



Thread A



Thread B

Synchronization

What went wrong?

The threads did not **communicate!**

- Same problem may occur with processes!

Some shared resource (array)

Time delay between checking some condition (loading the **tail**)

And the action (updating the **tail**)

- Preempted during this delay!

Scheduling can be random and preemption can happen at any time

Need some way to *synchronize* the threads

- Need help from the operating system

Mutex

MUTual EXclusion

A mutex is a lock that only one thread can acquire

All other threads attempting to access the resource protected by a *locked* mutex will be blocked

```
#include <pthread.h>
```

```
int pthread_mutex_init(pthread_mutex_t, NULL)
```

- Creates a new unlocked mutex

```
int pthread_mutex_lock(pthread_mutex_t*)
```

- Waits until it can lock the mutex

```
int pthread_mutex_unlock(pthread_mutex_t*)
```


- Unlocks the mutex

Fixing Race Conditions Animated

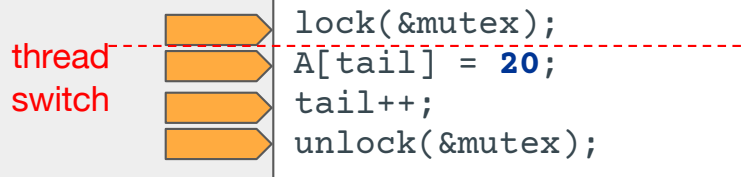
tail = 6

A[] =

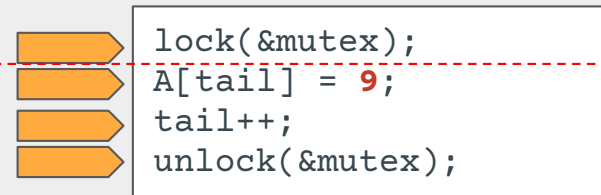
1	8	5	6	20	9		
---	---	---	---	----	---	--	--

mutex = 

enqueue()



Thread A



Thread B


Deadlocked

tail =

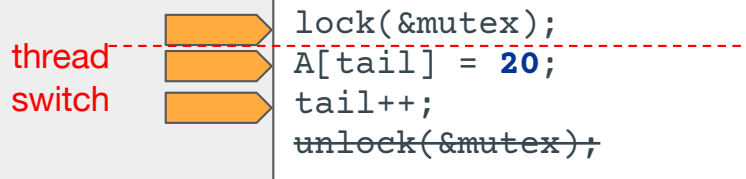
5

A[] =

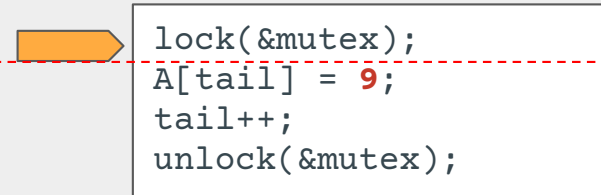
1	8	5	6	20			
---	---	---	---	----	--	--	--

mutex = 

enqueue()



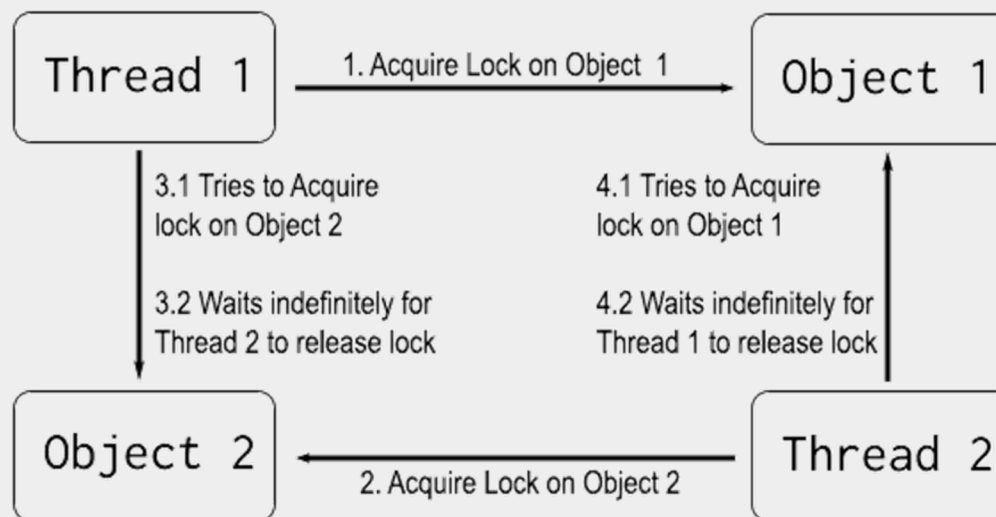
Thread A



Thread B
Never Runs!

Be careful with synchronization primitives

“A set of processes are **deadlocked** if each process in the set is waiting for an event only another process in the set can cause”



Semaphores

A special counter used for synchronization

- Essentially counts the number of *free resources*

Down (wait) reduces the counter

- Denoting that a resource is being used
- Waits if the counter is 0

Up (signal) operation increases the counter

- Denoting that a resource is now free

```
#include <semaphore.h>
```

```
int sem_init(sem_t*, 0, unsigned int initial_value);
```

- Creates a semaphore with the given initial value. (The second argument means if the semaphore data is in shared memory. If non-zero, it can't be seen by other threads.)

```
int sem_wait(sem_t*);
```

- Decrements counter unless it is 0 in which case it waits.

```
int sem_post(sem_t*);
```

- Increments counter.

Quiz time!

Password is _____

Condition Variables

A condition under which a thread executes or is blocked

Condition Variables

Condition Variables are used to wait for a particular condition to become true

`wait(condition, lock)`: release lock, put thread to sleep until condition is signaled; when thread wakes up again, re-acquire lock before returning.

`signal(condition, lock)`: if any threads are waiting on condition, wake up one of them. Caller must hold lock, which must be the same as the lock used in the wait call.

`broadcast(condition, lock)`: same as signal, except wake up all waiting threads.

Condition Variables

Essentially a queue of waiting threads

Thread **B** waits for a signal on **CV** before running

- `wait(CV, ...);`

Thread **A** sends `signal()` on **CV** when time for **B** to run

- `signal(CV, ...);`

Condition Variables at the DMV

Consider PennDOT (DMV)

- Which serves two functions:
 1. Title work
 2. License renewal

Critical resource: representative; threads: people in line

When a title-works window representative comes to the window after a break, a condition 'title_window_ready' is satisfied.

The title representative could look for the next ticket (for title work) and **signal** the customer to come to the window.

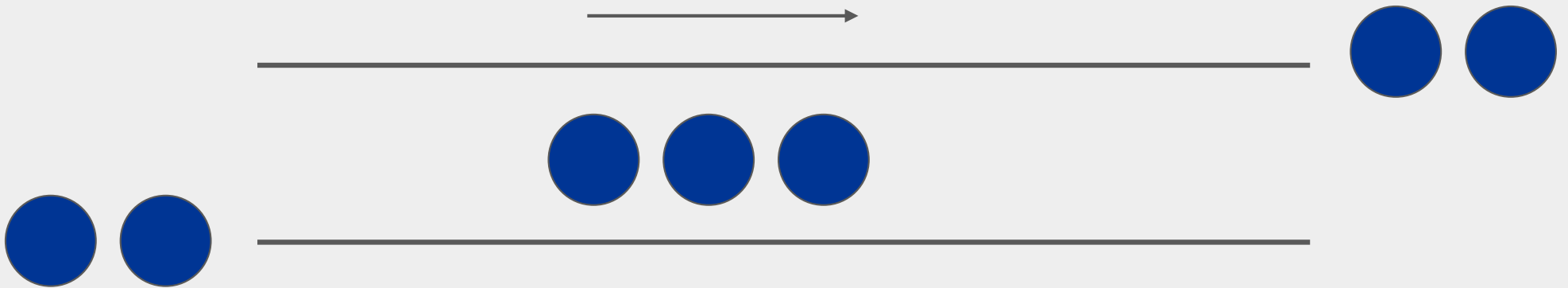
Here we have two condition variables, title_window_ready & license_window_ready. These conditions satisfy if one customer is handled and now the representative is ready to handle next customer.



The Bridge Problem

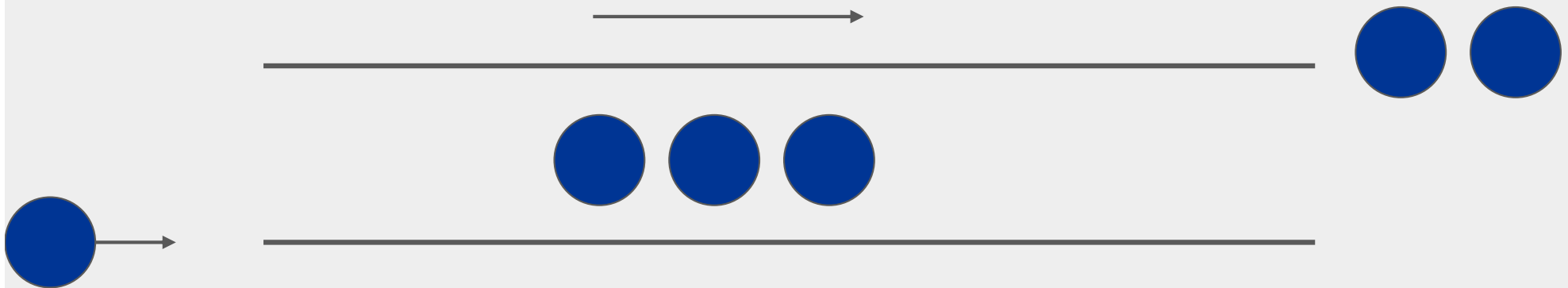
Consider a narrow bridge that can only allow three vehicles in the same direction to cross at the same time.

If there are three vehicles on the bridge, any incoming vehicle must wait as shown below.



The Bridge Problem

When new cars get to the bridge, have them wait



```
if(new car from left) wait(left, bridgelock)30
```

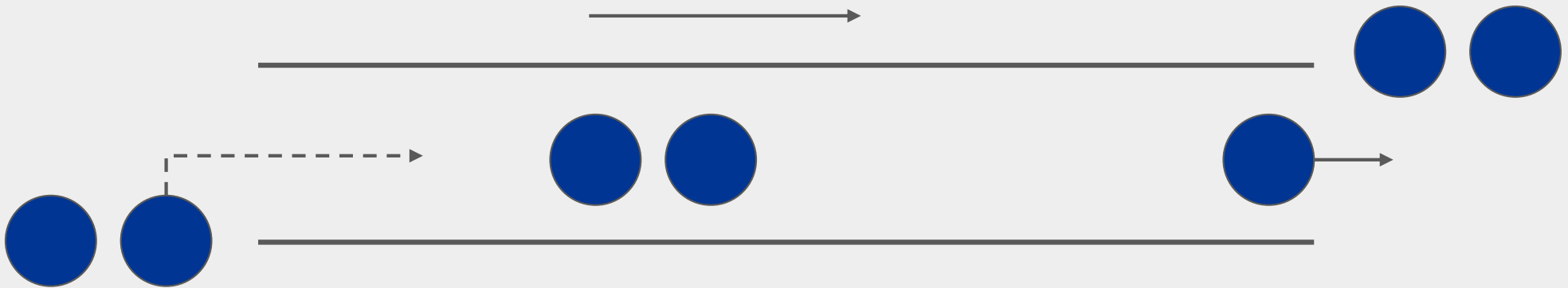
The Bridge Problem

When a vehicle exits the bridge, we have two cases to consider.

Case 1: there are other vehicles on the bridge

- Shown below
- In this case, one vehicle in the same direction should be allowed to proceed

Case 2: the exiting vehicle is the last one on bridge.



```
if(bridge.numCars != 0) signal(left, bridgelock)31
```

The Bridge Problem

Case 2 is more complicated and has two subcases.

In this case, the exiting vehicle is the last vehicle on the bridge.

If there are vehicles waiting in the opposite direction, one of them should be allowed to proceed. This is illustrated below:



```
if(bridge.numCars == 0 && rightCars != 0)
    signal(right, bridgelock)
```


The Bridge Problem

Or, if there is no vehicle waiting in the opposite direction, then let the waiting vehicle in the same direction to proceed.



```
if(bridge.numCars == 0 && rightCars == 0)  
    signal(left, bridgelock)
```

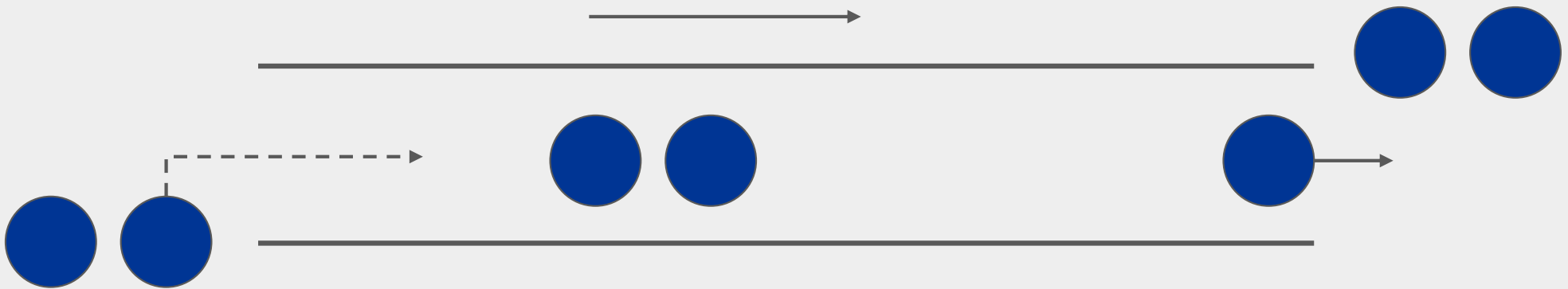
Problem with the Bridge Problem

Consider Case 1: there are other vehicles on the bridge

- Shown below
- In this case, one vehicle in the same direction should be allowed to proceed

But what if there are **infinite number of vehicles** on the left?

- Will the vehicles on the right ever get to go?



```
if(bridge.numCars != 0) signal(left, bridgelock)34
```

Starvation

Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads.

Live Lock

A **Livelock** is when two tasks are actively signaling the other to go and making no progress.

Example: Two friends at a dinner table with only one spoon

- *A tells B to use the spoon and eat first*
- *B tells A to use the spoon and eat first*
- *A tells B to use the spoon and eat first*
- *...*
- *No one gets to eat*

Aside: This is a weird example...

- Why are you at a dinner table with only one spoon?
- Why doesn't one of them go and get another spoon?

Many 'classical IPC problems' are built around weird premises ¹⁵⁵⁰

- Dining Philosopher Problems
- Sleeping Barber Problem