Concurrency Control

POSIX

How about two new threads?

```c
void hello(int *n)
{
    printf("hello %d\n", *n);
}

int main(void)
{
    int m;
    pthread_t thr1, thr2;

    m = 1;
    pthread_create(&thr1, NULL, (void(*)(void*))hello, (void*)&m);
    m = 2;
    pthread_create(&thr2, NULL, (void(*)(void*))hello, (void*)&m);
    ...
}
```
This creates two threads, both running the same code, namely \texttt{hello}, but on separate threads. Each thread has its own stack, thus its own copy of \texttt{n}.
This creates two threads, both running the same code, namely \texttt{hello}, but on separate threads. Each thread has its own stack, thus its own copy of \texttt{n}

Unfortunately, it is buggy code!
This creates two threads, both running the same code, namely `hello`, but on separate threads. Each thread has its own stack, thus its own copy of `n`.

Unfortunately, it is buggy code!

As usual, it may appear to run correctly several times, printing "hello 1" and "hello 2" (in either order!)
This creates two threads, both running the same code, namely \texttt{hello}, but on separate threads. Each thread has its own stack, thus its own copy of \texttt{n}.

Unfortunately, it is buggy code!

As usual, it may appear to run correctly several times, printing "\texttt{hello 1}" and "\texttt{hello 2}" (in either order!)

But sometimes it prints "\texttt{hello 2}" and "\texttt{hello 2}"
This is another case of sequential assumptions not following into parallel code: another race condition
This is another case of sequential assumptions not following into parallel code: another race condition

It *looks* like we update \( m \) in between the two new threads
This is another case of sequential assumptions not following into parallel code: another race condition.

It *looks* like we update $m$ in between the two new threads.

But the new threads are in parallel, running *asynchronously* with the main thread.
What we expect is

<table>
<thead>
<tr>
<th>main</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>creates 1</td>
<td>1 starts running</td>
<td>2 starts running</td>
</tr>
<tr>
<td>updates m</td>
<td>reads m=1</td>
<td>reads m=2</td>
</tr>
<tr>
<td>creates 2</td>
<td>prints 1</td>
<td>prints 2</td>
</tr>
</tbody>
</table>
What might happen is

<table>
<thead>
<tr>
<th>main</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>creates 1</td>
<td>1 starts running</td>
<td>2 starts running</td>
</tr>
<tr>
<td>updates m</td>
<td>reads m=2</td>
<td>reads m=2</td>
</tr>
<tr>
<td>creates 2</td>
<td>prints 2</td>
<td>prints 2</td>
</tr>
</tbody>
</table>

If thread 1 starts running slightly later
What might happen is

<table>
<thead>
<tr>
<th>main</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>creates 1</td>
<td>1 starts running</td>
<td>2 starts running</td>
</tr>
<tr>
<td>updates m</td>
<td>reads m=2</td>
<td>reads m=2</td>
</tr>
<tr>
<td>creates 2</td>
<td>prints 2</td>
<td>prints 2</td>
</tr>
</tbody>
</table>

If thread 1 starts running slightly later

In fact, this is quite likely, as creating a new thread takes a fair amount of time
There are three threads in the program: the two running `hello` and the one running `main`
There are three threads in the program: the two running *hello* and the one running *main*

The threads are *sharing* the variable *m* (via the pointers), so the behaviour of the program is dependent on what order the threads happen to access *m*. This is again bad programming, a data race
There are three threads in the program: the two running `hello` and the one running `main`.

The threads are *sharing* the variable `m` (via the pointers), so the behaviour of the program is dependent on what order the threads happen to access `m`. This is again bad programming, a data race.

Be very careful about the values you pass into the thread.
We can fix that race by not sharing:

```c
void hello(int *n)
{
    printf("hello %d\n", *n);
}

int main(void)
{
    int m1, m2;
    pthread_t thr1, thr2;

    m1 = 1;
    pthread_create(&thr1, NULL, (void(*)(void*))hello, (void*)&m1);
    m2 = 2;
    pthread_create(&thr2, NULL, (void(*)(void*))hello, (void*)&m2);

    return 0;
}
```
But now we (still) have another race condition, which fortunately is easier to spot.
Concurrency Control
POSIX

But now we (still) have another race condition, which fortunately is is easier to spot

We *might* see both hellos, but more likely is we will see nothing at all
But now we (still) have another race condition, which fortunately is is easier to spot

We *might* see both hellos, but more likely is we will see nothing at all

Again, the main thread *continues to run* and might return before the new threads have had chance to get started
But now we (still) have another race condition, which fortunately is is easier to spot

We might see both hellos, but more likely is we will see nothing at all

Again, the main thread continues to run and might return before the new threads have had chance to get started

In C, when the main function returns the whole process exits, and all of the threads are terminated, possibly before they have had chance to print
To fix this the initial thread should wait for the other threads to finish.
To fix this the initial thread should wait for the other threads to finish

```c
int pthread_join(pthread_t thread, void **retval);
```

This blocks the calling thread until the named thread exits.
To fix this the initial thread should wait for the other threads to finish

```c
int pthread_join(pthread_t thread, void **retval);
```

This blocks the calling thread until the named thread exits

This is the main use of the thread identifiers: joining threads (waiting for threads to finish)
To fix this the initial thread should wait for the other threads to finish

```c
int pthread_join(pthread_t thread, void **retval);
```

This blocks the calling thread until the named thread exits

This is the main use of the thread identifiers: joining threads (waiting for threads to finish)

A thread can end by returning from its initial function or by calling

```c
 pthread_exit(void *retval);
```
The thread can return a value, which is a pointer. This will be copied into where retval in pthread_join points.
The thread can return a value, which is a pointer. This will be copied into where `retval` in `pthread_join` points.

Use `NULL` if you don’t need a return value.
The thread can return a value, which is a pointer. This will be copied into where retval in pthread_join points.

Use NULL if you don’t need a return value.

Be careful not to return a pointer to something on the stack of the exiting thread!
The thread can return a value, which is a pointer. This will be copied into where retval in pthread_join points.

Use NULL if you don’t need a return value.

Be careful not to return a pointer to something on the stack of the exiting thread!

Any thread can wait for any other thread to terminate, as long as it knows the thread’s id (the pthread_t).
int main(void)
{
    int m1, m2;
    pthread_t thr1, thr2;

    m1 = 1;
    pthread_create(&thr1, NULL, (void(*)(void*))hello, (void*)&m1);
    m2 = 2;
    pthread_create(&thr2, NULL, (void(*)(void*))hello, (void*)&m2);
    pthread_join(thr1, NULL);
    pthread_join(thr2, NULL);
    return 0;
}
Concurrency Control

POSIX

- If any thread calls `exit()` anywhere, the entire process dies
Concurrency Control

POSIX

• If any thread calls `exit()` anywhere, the entire process dies
• If any thread calls `pthread_exit()` anywhere, that thread dies
• There is no hierarchy of threads, all threads are equal and independent once created

The only thing to watch out for is the thread running `main()`, because in C the `main()` function has an implicit `exit()` after its end. So if it finishes, the entire process subsequently dies.
Concurrent Control

POSIX

- If any thread calls `exit()` anywhere, the entire process dies
- If any thread calls `pthread_exit()` anywhere, that thread dies
- If any thread returns from its initial function, that thread dies
Concurrency Control

POSIX

- If any thread calls `exit()` anywhere, the entire process dies
- If any thread calls `pthread_exit()` anywhere, that thread dies
- If any thread returns from its initial function, that thread dies
- There is no hierarchy of threads, all threads are equal and independent once created

The only thing to watch out for is the thread running `main`, because in C the `main()` function has an implicit `exit()` after its end. So if it finishes, the entire process subsequently dies.
Concurrency Control

POSIX

- If any thread calls `exit()` anywhere, the entire process dies
- If any thread calls `pthread_exit()` anywhere, that thread dies
- If any thread returns from its initial function, that thread dies
- There is no hierarchy of threads, all threads are equal and independent once created

The only thing to watch out for is the thread running `main`, because in C the `main()` function has an implicit `exit()` after its end. So if it finishes, the entire process subsequently dies
Exercise. Think about what coding would be needed if we wanted always to get `hello 1` printed first and `hello 2` second

Exercise. Then generalise to $n$ threads
Advanced Exercise. The following code might cause a segmentation violation. Why?

```c
int main(void)
{
    int m1, m2;
    pthread_t thr1, thr2;

    m1 = 1;
    pthread_create(&thr1, NULL, (void(*)(void*))hello, (void*)&m1);
    m2 = 2;
    pthread_create(&thr2, NULL, (void(*)(void*))hello, (void*)&m2);
    return 0;
}
```