Concurrency:
Multi-core Programming

& Data Processing

Lab 10

-- Barrier Implementation --

Classical one

- Delays the execution of the threads until all of them reach a certain point
  - e.g., until all of them started
- Simplest approach:

```
public static AtomicInteger mySyncPoint = new
AtomicInteger(0);  // shared by all threads
public void run() {
   ...
   mySyncPoint.addAndGet(1);
   while (mySyncPoint.get() < numberOfThreads) {};
   ...
```

Complex one

- **Context:**

```
while (true) {
   CS.lock();
   CS
   CS.unlock();
   // desperately need barrier HERE
}
```

Complex one (cont'd)

- Why?... Hello Fairness, my old friend...
- In the face of a critical section, all threads must be treated equally!
- That is, no thread should be left behind while other pass through the critical section infinitely
- Threads have to pass through two doors
  - trapped between them, no thread can go further without the others
  - like some double-barriers in some parkings

Part I

```
B.lock();
passedthreads++;
if (passedthreads == N)
{
exitdoor = false;
entrancedoor = true;
}
B.unlock();
while
(!entrancedoor){};
```

Part II

```
B.lock();
passedthreads--;
if (passedthreads == 0)
{
entrancedoor = false;
exitdoor = true;
}
B.unlock();
while (!exitdoor){};
```

Exercise

- Explain how the complex barrier functions. Give an example trace on at least three threads. Why are the two consecutive parts needed? Would only Part I be enough to enforce the requirements? Explain.