# STDISCM - Problem Set 3

## Networked Producer and Consumer

by Lord John Benedict Pinpin
and Brett Harley Mider

# Program implementation

- The consumer (GUI window) is hosted on the local machine (port 8081)
- The producer is hosted in a Docker container (port 8082)
- Consumer and Producer communicate via network sockets
  - 4096-byte chunks per loop
- All code is written in Java (and Java external libraries)
  - Consumer GUI window is implemented via Javalin
- Consumer's GUI window and Producer's Docker instance are built and launched automatically by Main
- Parameters for producer threads (p), consumer threads (c), and max queue length (q) are obtained from the user at program start via command line
- Videos are uploaded from **`videos`**, downloaded in **`storage`**

# Queueing

- Producer and Consumer both use Java's BlockingQueue
- Producer uses a `LinkedBlockingQueue` separate mutex locks (`ReentrantLock`) for read and write operations, which prevents race conditions between the Producer and Consumer
- Consumer uses a `ArrayBlockingQueue` which is a bounded queue, limiting the number of videos in the queue
    - Supports fairness policy to avoid starvation but is implemented as FIFO queue
- Videos (.mp4 files) are queued and gradually gets transferred from producer to consumer; if many videos are detected at once, the order they are queued is random
- Consumer drops videos if upload queue is full (leaky bucket design)
- Consumer does not queue if video duplicates are detected (done via hashing)

# Producer and consumer concepts applied

- Bounded buffer
  - Producer and consumer deals with a limited buffer, dropping videos if necessary
- Race conditions
  - Producer and consumer shares data (video queue) which they can concurrently access
- Mutual exclusion
  - Done so that producer and consumer do not access the queue at the same time which leads to race conditions
- Deadlock avoidance
  - Done so that the producer and consumer don’t wait indefinitely for videos to appear in the source directory and the video queue, respectively
- Blocking queue
  - Consumer can only deal with limited videos at a time, dropping videos if the queue is full instead of waiting to handle them

# Synchronization mechanisms used

- Mutex Lock
  - Ensures only one thread at a time accesses the critical section (shared buffer/video queue)
- Condition variables
  - Blocks threads from executing code by waiting a certain condition (e.g. video is queued, video is full)
- Java's `BlockingQueue`
  - Abstracts and implements both mutex locks and condition variables, integrating them seamlessly with a thread-safe, synchronized data structure