CS340
Lecturer: Dr. Simina Fluture

Read: **Synchronization Problems**

Topics: The Bounded - Buffer Problem (producer - consumer)
The Readers-Writers Problem
The Dining-Philosophers Problem

The Bounded-Buffer Problem

Consider a system with two types of threads:

- A producer of information.
- A consumer that uses the information produced.

A buffer with size of ***N*** is allocated to the pair of processes for information exchange. The system is charged with preventing the overlap of buffer operations.
Considering the above description of the Producer-Consumer problem, implement (using pseudo-code) semaphores that will synchronize these threads in their operations on shared data structures.

1: Identify the involved threads and (without synchronizing them yet) sketch their execution pseudo-code:

```
Producer                                        Consumer
  repeatedly                                      repeatedly
    produce item;                                     wait until a full buffer exists;
    wait until an empty buffer is available;          get a full buffer;
    get an empty buffer;                              consume (read) the item;
    fill the buffer;
  until false;                                      until false;
```

2: Identify what are the shared variables (structures) and Critical Sections.
Decide the number, type and initial values of needed semaphores.

The OS allows some reasonable FIXED AMOUNT of storage for buffering communications. The consumer removes data in THE ORDER in which it was deposited.

3: Give the execution pseudo-code that will implement the synchronization of processes (threads).
Explain (your) solution. Discuss (if it is the case) any possible drawbacks.

Possible exam questions: (covered in class)

Java code: **BoundedBuffer.java , BoundedBufferServer.java, Consumer.java, Producer.java**

The Readers-Writers Problem

Suppose that a resource (data object-file, block of main memory, bank of processor registers etc.) is to be shared among a set of concurrent processes of two distinct types: **readers** and **writers**. Readers only read the shared resource (they are non-destructive). Writers update data (modify).

A process is **active** if it is currently reading or writing or it is in the CS that allows it to do so.
A process is **waiting** if it is attempting to enter the CS (performs a wait operation).
A process is **arriving** just at the point that it is about to try (attempt) to enter the CS.

Several policies could be implemented for managing the shared resource, all involving priorities.
First Reader-Writer Policy: an *Arriving* writer waits until there are no *Active* readers (no reader will be kept waiting unless a writer has already obtained permission to use the shared resource).
Second Reader-Writer Policy: if a writer is Waiting to access the shared object, no *Arriving* readers may start reading (if a writer is waiting to access the object, no new reader may start reading).
The first policy can result in starvation of the writers, the second policy can result in starvation of the readers.

First Reader-Writer Policy

An *Arriving* writer waits until there are no *Active* readers (no reader will be kept waiting unless a writer has already obtained permission to use the shared resource.)
It is necessary for the first reader to access the shared resource to compete with any writers, but any succeeding readers may pass directly into the critical section, provided that a reader is still there.

Reader

```
while (true) {
if there is Writer in the DB        // startRead( )
wait .........                      // entry sec

DB                                  // access DB

finish reading                      // endRead( )
                                    // exit sec
}
```

Writer

```
while(true) {
if DB is Not free                   // startWrite( )
wait..........                      // entry sec

DB                                  // access DB

finish writing                      // endWrite( )
                                    // exit sec
}
```

The readers keep track of the number of readers in the critical section through the **readCount** variable. The checking or modification of the readCount has to be done exclusively by only one reader at a time. It represents the **Critical Section**.
Only the first reader executes the wait(OKaccessDB) operation, while every writer does so. The last reader to yield the critical section must perform the signal operation for all readers that were accessing the shared resource.

```
        resource_type  Database;
        int readerCount =
        bin sem mutex =
        bin sem OKaccesDB =
```

Using the above specifications write the pseudocode for the Reader and the Writer in your notes.
OBS: readers can dominate the resource so that no writer ever gets a chance to access it. In real systems it is analogous to the case in which a pending update of a file must wait until all reads have been completed.
Open questions: (covered in class)

The Dining-Philosophers Problem

The Dining-philosophers problem is a representation of the need to allocate several resources between several processes in a deadlock and starvation free manner.
Philosophers share a common circular table surrounded by five chairs, each belonging to one philosopher. The table is laid with five single chopsticks. When a philosopher thinks, he does not interact with the other philosophers. From time to time a philosopher gets hungry and tries to pick up the two chopsticks that are closest to him. A philosopher may pick up only one chopstick at a time. When a hungry philosopher has two chopsticks at the same time, he eats. When he is finished eating, he puts down both of the chopsticks and starts thinking.

One solution is by representing each chopstick by a semaphore.
Each Philosopher will execute:

```
While (true) {
    Think               // and getting hungry
    Get left chopstick
    Get right chopstick
    Eat
    Putdown left chopstick
    Putdown right chopstick
}
```