6G7Z1004 Advanced Computer Networks and Operating Systems
(Lab 5: Concurrency – Semaphores & Deadlock)
This lab was originally designed by Prof. Rene Doursat, MMU, UK

In this lab session, we will continue looking at the race conditions problem; perform deep analysis of a mutual exclusion solution using "semaphores" and use "Resource Allocation Graphs: RAG" to identify "Deadlock" situations.

Task A: Synchronization with Semaphores (see the slides 49-50 of Concurrency lecture)

```
item[3] buffer;    // initially empty
semaphore empty;   // initialized to +3
semaphore full;    // initialized to 0
binary_semaphore mutex;    // initialized to 1
```

```
void producer()
{
    while (true) {
        item = produce();

p1:     wait(empty);
   /    wait(mutex);
p2:     append(item);
   \    signal(mutex);
p3:     signal(full);
    }
}
```

```
void consumer()
{
    while (true) {
c1:     wait(full);
   /    wait(mutex);
c2:     item = take();
   \    signal(mutex);
c3:     signal(empty);

        consume(item);
    }
}
```

The above pseudocode is a correct implementation of the producer/consumer problem with a bounded buffer. Labels p1, p2, p3 and c1, c2, c3 refer to the lines of code shown above (p2 and c2 cover two lines of code). Semaphores **empty** and **full** are integer semaphores that can take unbounded negative and positive values. There are multiple producer processes, referred to as Pa, Pb, Pc, etc., and multiple consumer processes, referred to as Ca, Cb, Cc, etc. Each semaphore maintains a FIFO (first-in-first-out) queue of blocked processes. In the scheduling chart below, each line represents the state of the buffer and semaphores after the scheduled execution has occurred. To simplify things, we assume that scheduling is such that processes are never interrupted while executing a given portion of code p1, or p2, …, or c3. Your task is to complete the chart in the next page.

Scheduled step of execution	`full`'s state & queue	Buffer	`empty`'s state & queue
Initialization:	full = 0	_ _ _	empty = +3
Ca executes c1:	full = -1 (Ca)	_ _ _	empty = +3
Cb executes c1:	full = -2 (Ca, Cb)	_ _ _	empty = +3
Pa executes p1:	full = -2 (Ca, Cb)	_ _ _	empty = +2
Pa executes p2:	full = -2 (Ca, Cb)	X _ _	empty = +2
Pa executes p3:	full = -1 (Cb) Ca is freed	X _ _	empty = +2
Ca executes c2:	full = -1 (Cb)	_ _ _	empty = +2
Ca executes c3:	full = -1 (Cb)	_ _ _	empty = +3
Pa executes p1:	full =		empty =
Pa executes __:	full =		empty =
Pb executes p1:	full =		empty =
Pc executes p1:	full =		empty =
Pb executes __:	full =		empty =
Pb executes __:	full =		empty =
Cb executes __:	full =		empty =
Pa executes __:	full =		empty =
Pc executes __:	full =		empty =
Cb executes __:	full =		empty =
Ca executes c1-c3:	full =		empty =
Pc executes __:	full =		empty =
Pb executes p1:	full =		empty =
Pa executes p1-p3:	full =		empty =
Pd executes p1:	full =		empty =
Pe executes __:	full =		empty =
Cc executes c1-c3:	full =		empty =
Cd executes c1-c3:	full =		empty =
Pe executes p2-p3:	full =		empty =
Pd executes p2-p3:	full =		empty =
Pb executes __-__:	full =		empty =

Task B: Deadlocks (see the slides 54-56 of Concurrency lecture)

In the pseudocode below, three processes, P0, P1 and P2, are competing for six resources labeled A to F:

```
void P0()
{
  while (true) {
    get(A);
    get(B);
    get(C);
    // critical section:
    // use A, B, C
    release(A);
    release(B);
    release(C);
  }
}
```

```
void P1()
{
  while (true) {
    get(D);
    get(E);
    get(B);
    // critical section:
    // use D, E, B
    release(D);
    release(E);
    release(B);
  }
}
```

```
void P2()
{
  while (true) {
    get(C);
    get(F);
    get(D);
    // critical section:
    // use C, F, D
    release(C);
    release(F);
    release(D);
  }
}
```

a) Show the possibility of a deadlock in this implementation by drawing a Resource Allocation Graph (RAG).

b) Modify the order of some of the get() requests to prevent the possibility of any deadlock. You cannot exchange requests between procedures; you can only change the order inside each procedure. Draw another resource allocation graph to justify your answer.