1. Synchronization with semaphores

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 = -1 (Cb)	_ _ _	empty = +2
Pa executes p2:	full = -1 (Cb)	X _ _	empty = +2
Pb executes p1:	full = -1 (Cb)	X _ _	empty = +1
Pc executes p1:	full = -1 (Cb)	X _ _	empty = 0
Pb executes p2:	full = -1 (Cb)	X X _	empty = 0
Pb executes p3:	full = 0 Cb is freed	X X _	empty = 0
Cb executes c2:	full = 0	X _ _	empty = 0
Pa executes p3:	full = +1	X _ _	empty = 0
Pc executes p2:	full = +1	X X _	empty = 0
Cb executes c3:	full = +1	X X _	empty = +1
Ca executes c1-c3:	full = 0	X _ _	empty = +2
Pc executes p3:	full = +1	X _ _	empty = +2
Pb executes p1:	full = +1	X _ _	empty = +1
Pa executes p1-p3:	full = +2	X X _	empty = 0
Pd executes p1:	full = +2	X X _	empty = -1 (Pd)
Pe executes p1:	full = +2	X X _	empty = -2 **(Pd, Pe)**
Cc executes c1-c3:	full = +1	X _ _	empty = -1 **Pd (Pe)**
Cd executes c1-c3:	full = 0	_ _ _	empty = 0 **Pd, Pe**
Pe executes p2-p3:	full = +1	X _ _	empty = 0 Pd
Pd executes p2-p3:	full = +2	X X _	empty = 0
Pb executes p2-p3:	full = +3	X X X	empty = 0

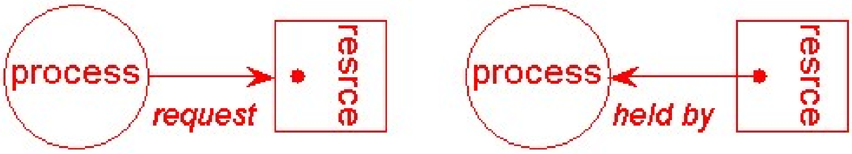
2. Deadlocks

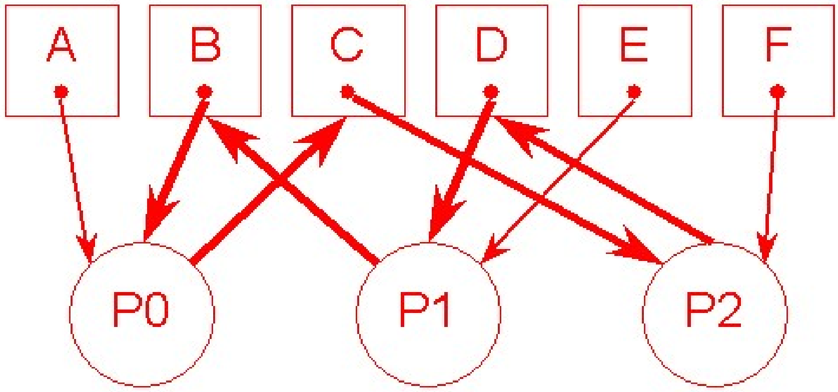
a) There is a deadlock if the scheduler goes, for example: P0-P1-P2-P0-P1-P2 (line by line): Each of the 6 resources will then be held by one process, so all 3 processes are now blocked at their third line inside the loop, waiting for a resource that another process holds. This is illustrated by the circular wait (thick arrows) in the RAG below: P0→C→P2→D→P1→B→P0.

```
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);
  }
}
```



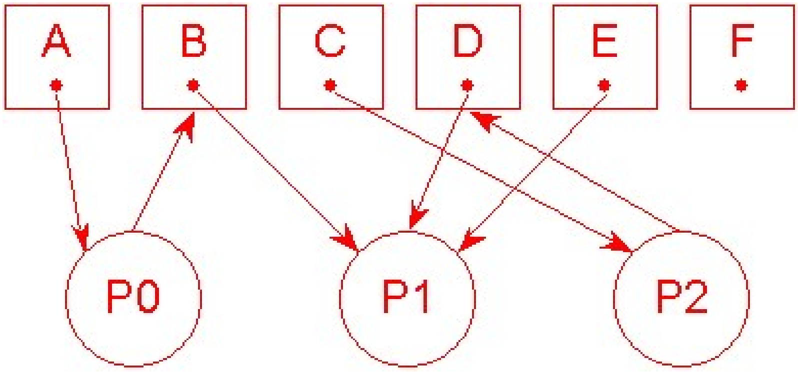


b) Any change in the order of the `get()` calls that alphabetizes the resources inside each process code will avoid deadlocks. More generally, it can be a direct or reverse alphabet order, or any arbitrary but predefined ordered list of the resources that should be respected inside each process.

Explanation: if resources are uniquely ordered, cycles are not possible anymore because a process cannot hold a resource that comes after another resource it is holding in the ordered list (see this remark in the section about Circular Wait Prevention, Stallings' book). For example:

get(A)	get(B)	get(C)
get(B)	get(D)	get(D)
get(C)	get(E)	get(F)

With this code, and starting with the same worst-case scheduling scenario P0-P1-P2, we can only continue with either P1-P1-CS1… or P2-P2-CS2…. For example, in the case P1-P1, we get the following RAG without circular wait:



After entering CS1, P1 then releases all its resources and P0 and P2 are free to go. Generally, the same thing would happen with any fixed ordering of the resources: one of the three processes will always be able to enter its critical area and, upon exit, let the other two progress.