1. When a process has requested some resource and that resource is busy, "busy wait" can be used to continuously check if the resource has been released.
While busy waiting consists of running an empty loop polling for the resource to be ready from the waiting process, non-busy waiting can be achieved using a queue.

This way it isn't using up CPU cycles just checking the state of the resource. Busy waiting has little to no overhead when the expected waiting time is low, but isn't a pattern to use for larger processes due to the consummation of CPU cycles.

2. If wait() is not done atomically, then it is possible for multiple processes to enter the same critical (mutual exclusion violated). This happens if processes overlap in a certain way.

> Thread A checks the semaphore value and sees that it is 1, i.e. larger than 0, so it is about to decrement the semaphore, but is interrupted by…
> a context switch to Thread B, which also sees that the semaphore value is 1, so it decrements it to 0 and enters its CS.
> A context switch back to A leads to the decrementation of 0 to -1 (or something), and A enters its CS.

If signal is not done atomically, then, in a similar way, the available resources may be misrepresented and the system may allocate resources it doesn't have, leading to processes entering the CS at the same time.

3.

a) Need = Max - Allocation

Thread	Allocation	Max	Available	Need
T_0	0 0 1 2	0 0 1 2	1 5 2 0	0 0 0 0
T_1	1 0 0 0	1 7 5 0	1 5 2 0	0 7 5 0
T_2	1 3 5 4	2 3 5 6	1 5 2 0	1 0 0 2
T_3	0 6 3 2	0 6 5 2	1 5 2 0	0 0 2 0
T_4	0 0 1 5	0 6 5 6	1 11 5 2	0 6 4 2

b) Yes, a safe sequence exists (T_0, T_3, T_1, T_4, T_2)
c) Yes, the request is less than/equal to the need of T_1, and there are enough resources, so it can be granted immediately.

4.

a) yes, example: T_0 holds (1, 2, 1) and requests (1, 0, 0),T_1 holds (1, 0, 1) and requests (0, 2, 0), T_2 holds (2, 0, 0) and requests (0, 1, 1). Now, T_0 is waiting for a resource held by T_1, T_1 is waiting for a resource held by T_2, and T_2 is waiting

for a resource held by T_0. This creates a circular wait in the resource graph, causing a deadlock

b) Yes, indefinite blocking is possible, as the system prioritises active requests over blocked threads. A blocked thread might continually have its resources taken away to satisfy active requests, causing it to remain blocked. Since there is no fairness mechanism to ensure all threads eventually get the resources they need, a thread could get unlucky and keep having its resources taken away before it can use them.