the issues in before Peterson:

+add the values of the flag and change value of the flag=> two processes in a mutual pricession -> not synchronize

+

+a critical section: wait() get() immuted semaphore ==1

semaphore value ==0, suspend

it is not a good practice to do a wait() within a critical secrtion. Doing wait() release the critical sect

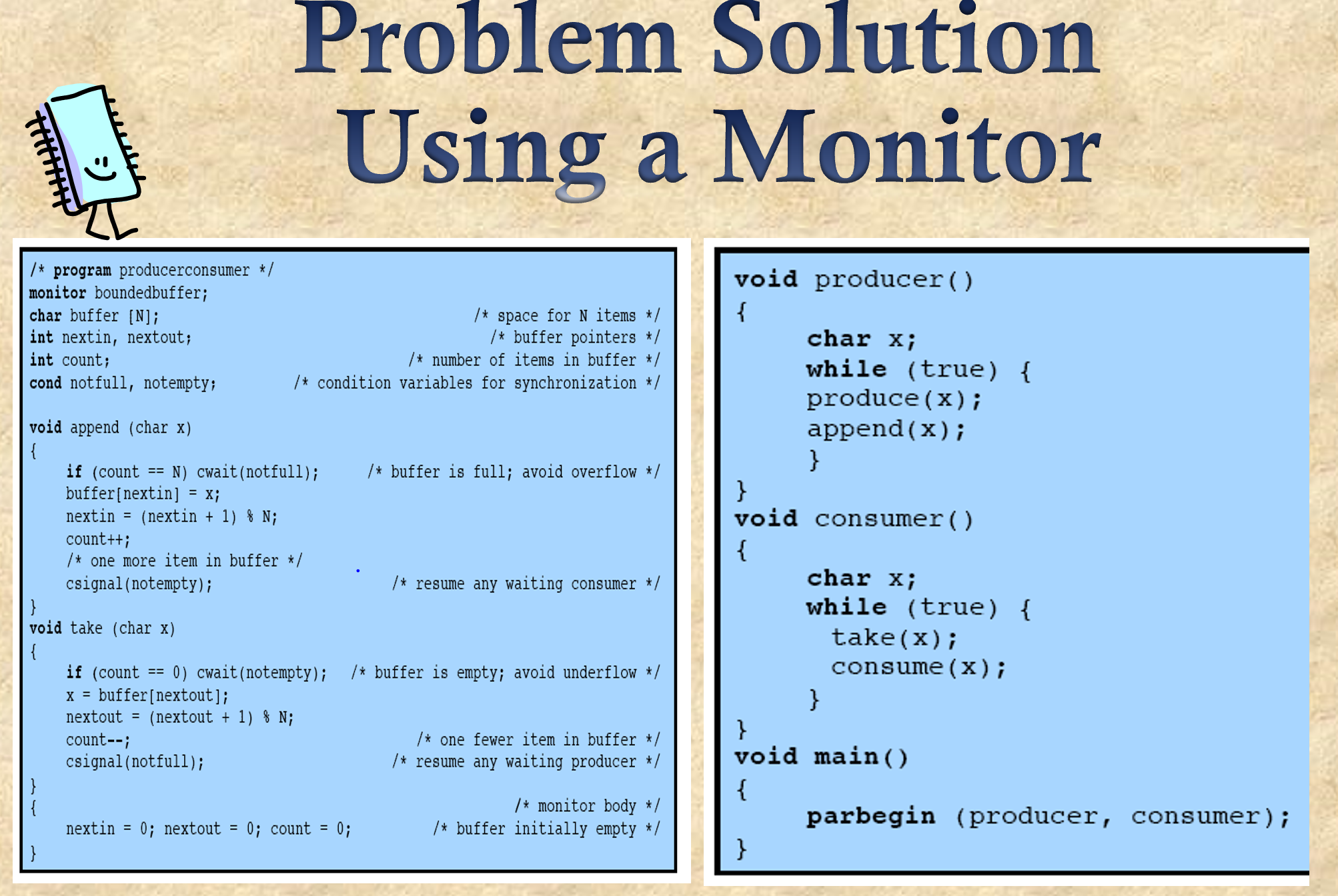
As an example of the use of a monitor, let us return to the bounded-buffer producer/consumer problem. Figure 5.16 shows a solution using a monitor. The monitor module, boundedbuffer , controls the buffer used to store and retrieve characters. The monitor includes two condition variables (declared with the construct **cond ): *notfull is true when there is room to add at least one character to the*** buffer, and *notempty is true when there is at least one character in the buffer.*

A producer can add characters to the buffer only by means of the procedure append inside the monitor; the producer does not have direct access to *buffer . The* procedure first checks the condition *notfull to determine if there is space available* in the buffer. If not, the process executing the monitor is blocked on that condition. Some other process (producer or consumer) may now enter the monitor. Later, when the buffer is no longer full, the blocked process may be removed from the queue, reactivated, and resume processing. After placing a character in the buffer, the process signals the *notempty condition. A similar description can be made of the* consumer function.

This example points out the division of responsibility with monitors compared to semaphores. In the case of monitors, the monitor construct itself enforces mutual exclusion: It is not possible for both a producer and a consumer simultaneously to access the buffer. However, the programmer must place the appropriate cwait and csignal primitives inside the monitor to prevent processes from depositing items in a full buffer or removing them from an empty one. In the case of semaphores, both mutual exclusion and synchronization are the responsibility of the programmer.

Note that in Figure 5.16 , a process exits the monitor immediately after executing the csignal function. If the csignal does not occur at the end of the procedure, then, in Hoare’s proposal, the process issuing the signal is blocked to make the monitor available and placed in a queue until the monitor is free. One possibility at this point would be to place the blocked process in the entrance queue, so that it would have to compete for access with other processes that had not yet entered the monitor. However, because a process blocked on a csignal function has already partially performed its task in the monitor, it makes sense to give this process precedence over newly entering processes by setting up a separate urgent queue ( Figure 5.15 ). One language that uses monitors, Concurrent Pascal, requires that csignal only appear as the last operation executed by a monitor procedure.

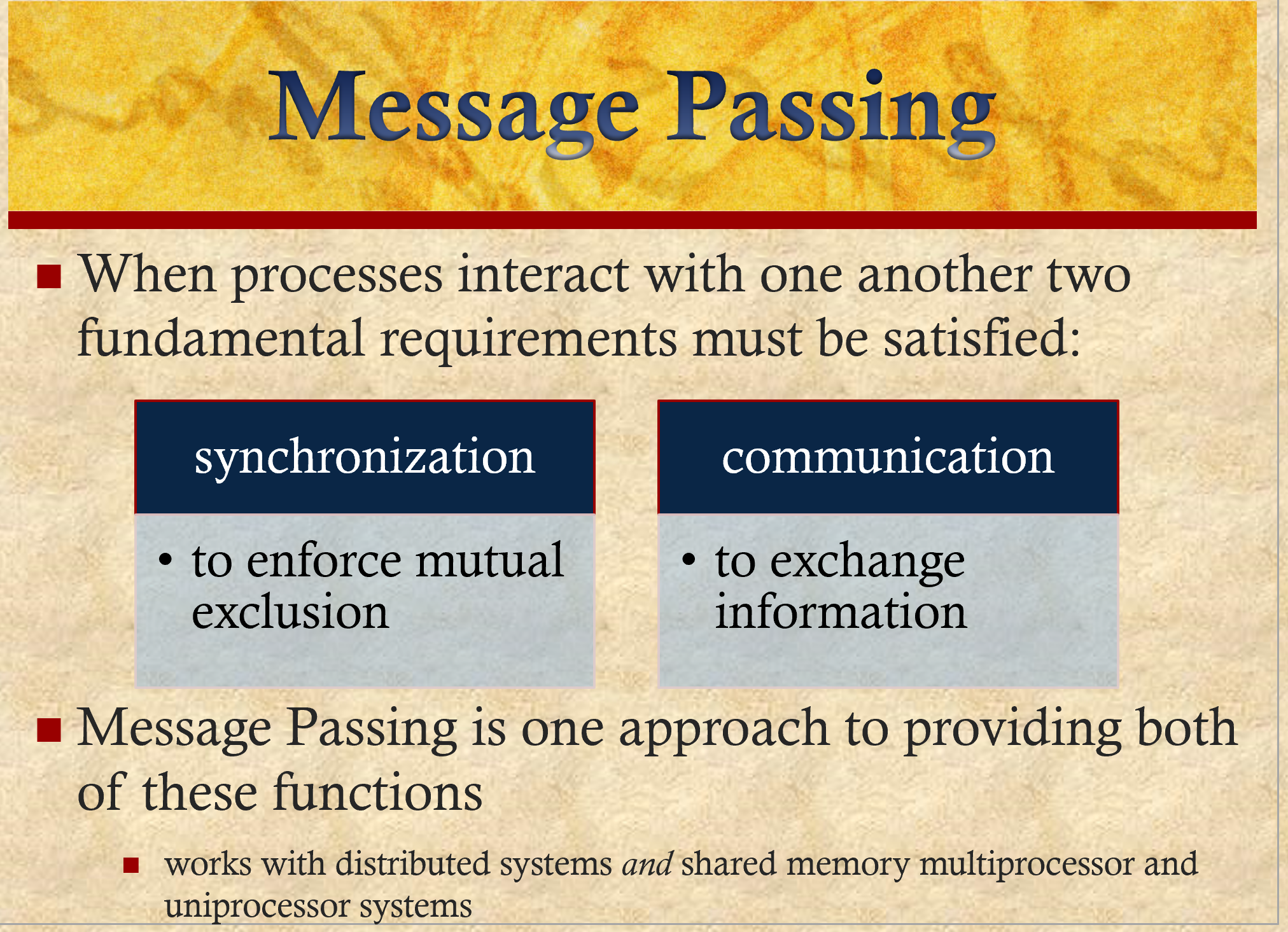
If there are no processes waiting on condition *x , then the execution of* csignal( *x ) has no effect.*

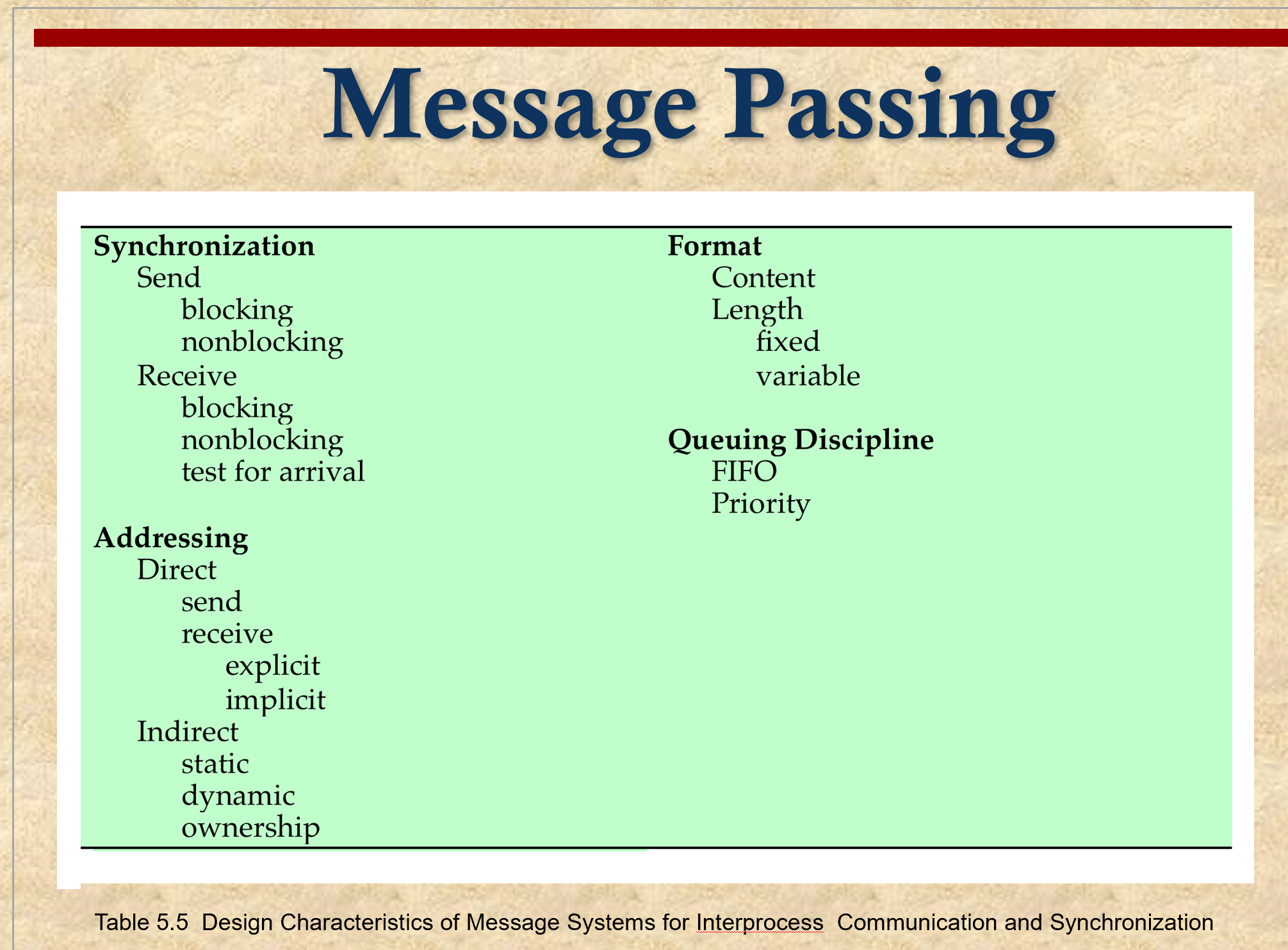


count: shared resource

monitor: java, pascal

There is no monitor in C++





guarantee mutual exclusion:

call out everything in buffer