Sample Questions for Autumn Mid Semester Examination 2024-25

Distributed Operating System (CS30009)

Topic: Blocking versus Non-blocking Primitives, Buffered versus Unbuffered Primitives

Short Question (1 mark)

Q1. Differentiate between blocking *receive* and non-blocking *receive* primitives.

Answer:

Blocking *receive()* primitive: when the receive statement is executed, the receiving process is halted until a message is received.
Non-blocking *receive()* primitive: The non-blocking *receive()* primitive implies that the receiving process is not blocked after executing the *receive()* statement, control is returned immediately after informing the kernel of the message buffer's location.

Q2. What is the issue that occurs in a non-blocking *receive()* primitive. How to fix it?
Answer:
The issue in a non-blocking *receive()* primitive is when a message arrives in the message buffer, how does the receiving process know?
One of the following two procedures can be used for this purpose:
Polling: In the polling method, the receiver can check the status of the buffer when a test primitive is passed in this method.
The receiver regularly polls the kernel to check whether the message is already in the buffer.
Interrupt: A software interrupt is used in the software interrupt method to inform the receiving process regarding the status of the message i.e. when the message has been stored into the buffer and is ready for usage by the receiver.

****

Long Question (2.5 marks)

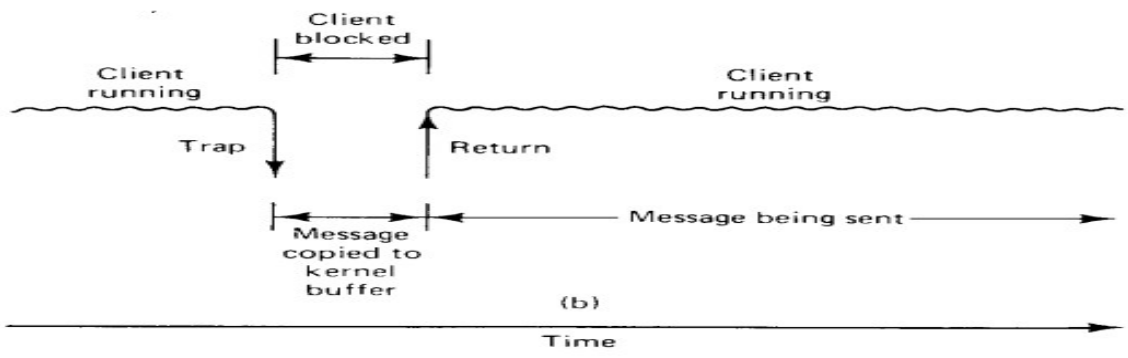
Q1. Explain non-blocking *send()* primitive in detail. What is the advantage and disadvantage of the non-blocking *send()* primitive? Explain the two possible solutions to overcome the disadvantage of the non-blocking *send()* primitive.

Answer:

- An alternative to blocking primitives are non-blocking primitives (also called asynchronous primitives).
- If *send* is non-blocking, it returns control to the caller immediately, before the message is sent.
- The advantage of this scheme is that the sending process can continue computing in parallel with the message transmission, instead of having the CPU go idle.
- The disadvantage of this scheme is that the sender cannot modify the message buffer until the message has been sent.
- The sending process has no idea of when the transmission is done, so it never knows when it is safe to reuse the buffer.

There are two possible solutions to this problem:

- The first solution is to have the kernel copy the message to an internal kernel buffer and then allow the process to continue, as shown in the figure shown below.
- The disadvantage of this method is that every outgoing message has to copied from user space to kernel space.

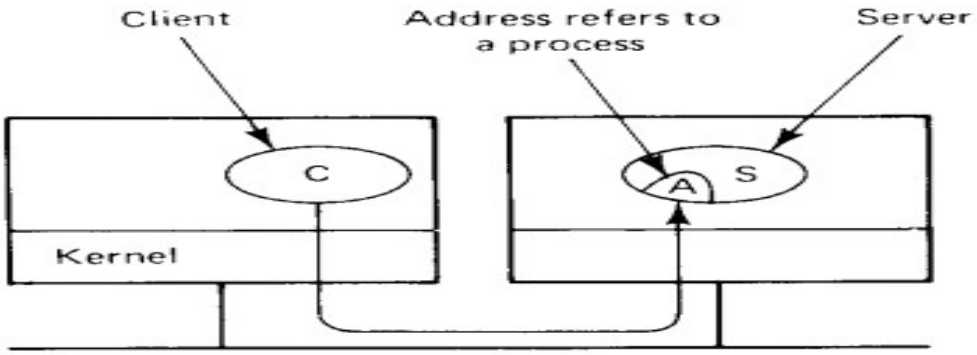


- The second solution is to interrupt the sender when the message has been sent to inform it that the buffer is once again available.
- No copy is required here, which saves time, but programs based on user-level interrupts are difficult to write and debug.

Q2. With the help of a diagram, explain unbuffered primitive in detail. What are the problems that occur when the client calls *send* primitive before the server calls *receive* primitive in an unbuffered message passing mechanism? Explain some strategies to handle these problems.

Answer:

- Unbuffered primitives involve direct communication without any intermediate storage.
- In these primitives, the sender and receiver need to be synchronized for the communication to take place.
- A call to the primitive *receive(addr, &m)* tells the kernel of the machine on which it is running that the calling process is listening to the address *addr* and is prepared to receive one message sent to that address.
- A single message buffer, pointed to by m, is provided to hold the incoming message.
- When the message comes in, the receiving kernel copies it to the buffer and unblocks the receiving process, as shown by the figure shown below.



The problems that occurs when the client calls send primitive before the server calls receive primitive in unbuffered message passing mechanism are:

How does the server's kernel know which of its processes is using the address in the newly arrived message?

How does the server's kernel know where to copy the message?

To avoid such problems, it's crucial to ensure that the receive primitive is called in a timely manner. Some strategies to handle this include:

Pre-emptive Design: Design the system so that the receive is always invoked before or concurrently with send to avoid blocking.

Timeouts and Error Handling: Implement timeouts or error handling mechanisms to manage situations where a send operation might block indefinitely.

Buffered Communication: Use buffered message passing where messages are stored in a buffer temporarily, allowing the sender and receiver to operate asynchronously and reducing the risk of blocking.

****