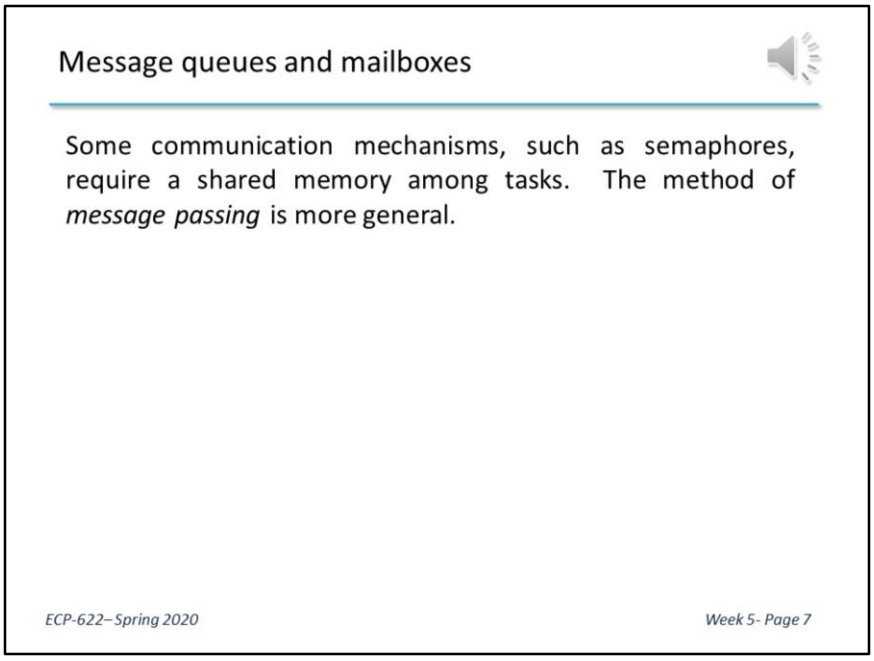


Another technique for information exchange and synchronization of tasks is the method of message passing. It is used in all types of operating systems. One of its advantages is that it can be used even if tasks run on different computers as in distributed systems. Note that the use of a semaphore assumes that all tasks have access to some memory location where semaphore is stored.

Message queues and mailboxes

Some communication mechanisms, such as semaphores, require a shared memory among tasks. The method of *message passing* is more general.

A minimum set of system calls that handle message passing is the following:

send (*destination*, &*message*)

receive (*source*, &*message*)

Two basic functions need to be provided by the system. The “send” function sends a message to another destination task. System takes a copy of message contents and put it in the memory space of the destination. The “receive” function requests the system to bring a message from some source. We will consider examples of real syntax later.

Message queues and mailboxes

Some communication mechanisms, such as semaphores, require a shared memory among tasks. The method of *message passing* is more general.

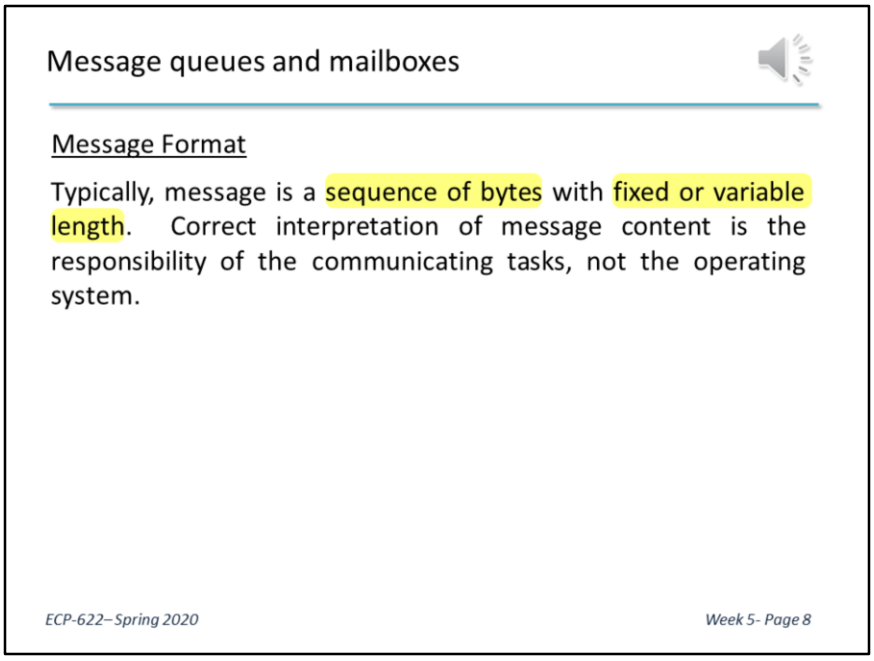
A minimum set of system calls that handle message passing is the following:

send (*destination*, &*message*)

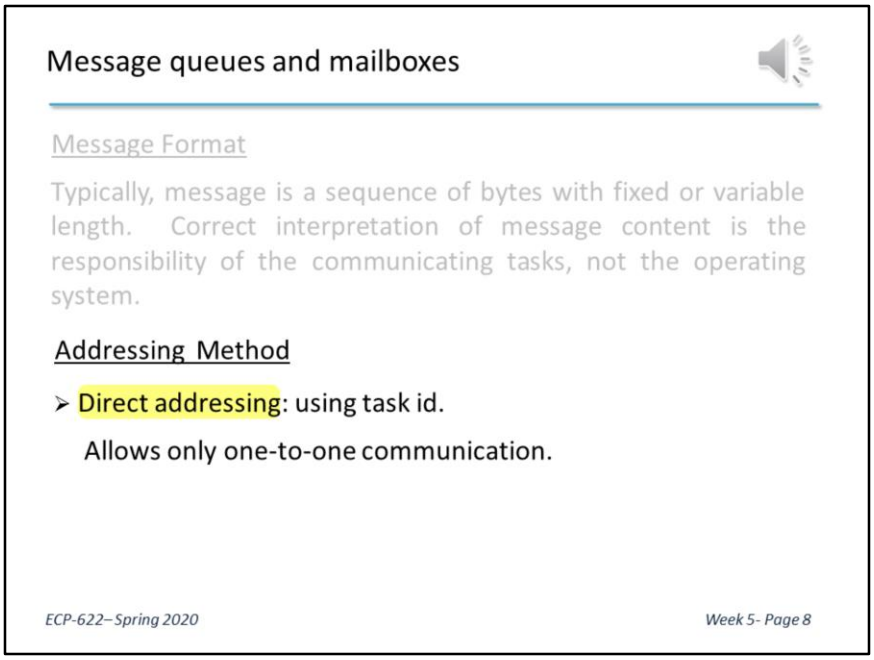
receive (*source*, &*message*)

Many design options exist for the message format, addressing methods, synchronization modes, and queuing discipline. All these can affect program timing.

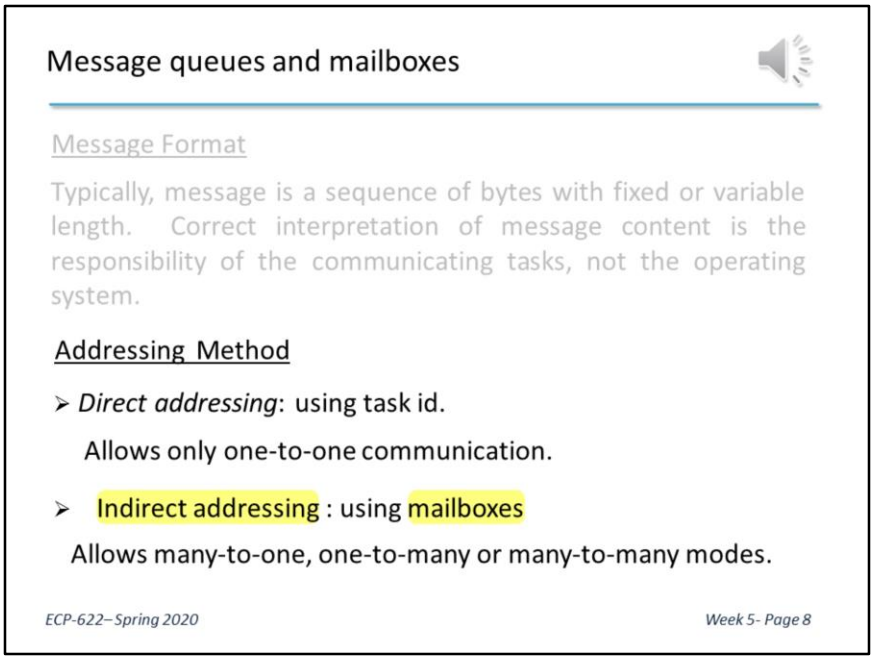
We next consider the many options that can be used in implementing the above two functions. According to size and complexity of operating system, it provides some subset of these options.



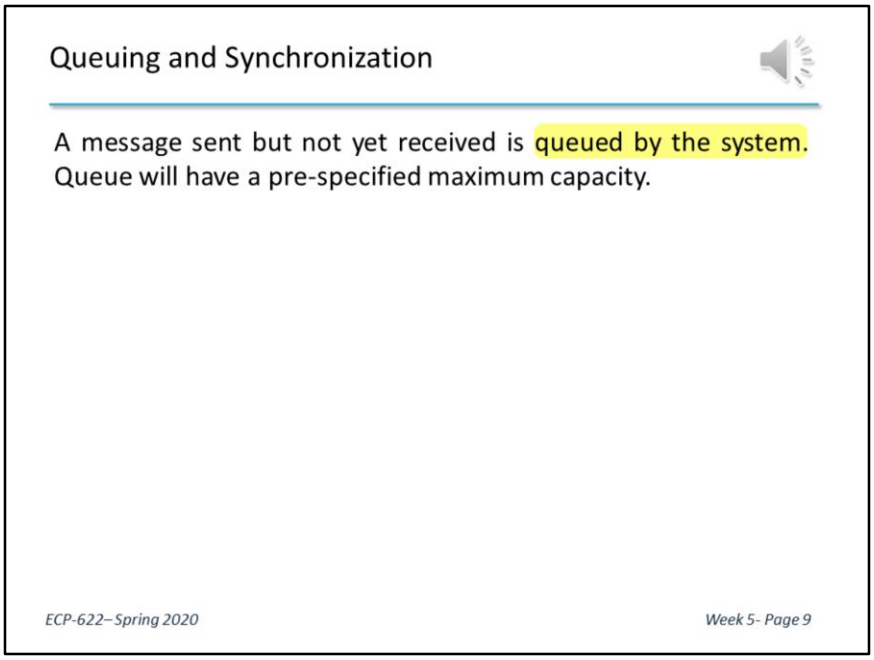
What can be the contents of message? Ideally, task can put any of its local variables in a message and sends it to another task. However, typically the same send function is used to send any data. From the system point of view, a sequence of bytes is sent regardless of its meaning. Fixed length messages simplify design (size of buffers, etc.) but longer messages may need to be segmented and shorter messages will need to be padded with extra bytes.



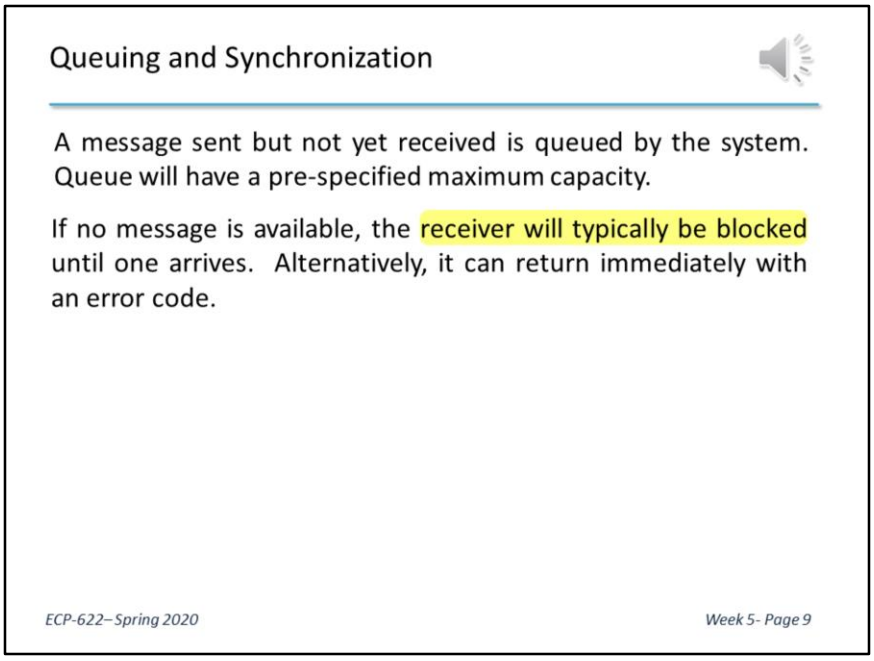
How to specify source and destination? In direct addressing, the task id is used to specify them. This allows only one task to exchange data with only one other task, and these id's should be known and fixed in the task codes.



To allow receiving a message from a number of possible sources, and send a message that can be received by several processes, the idea of mailbox is used. The mailbox is a buffer of messages that a task can ask the system to create. Then, any number of tasks can send messages to the mailbox, and any number of tasks can read messages from the mailbox. This is useful, for example for a server task that can receive a service request from an arbitrary task at any time.



System will need to maintain a queue at a task or mailbox to hold messages sent but not yet received. To avoid unbounded growth of this queue, its maximum length need to be specified at creation.



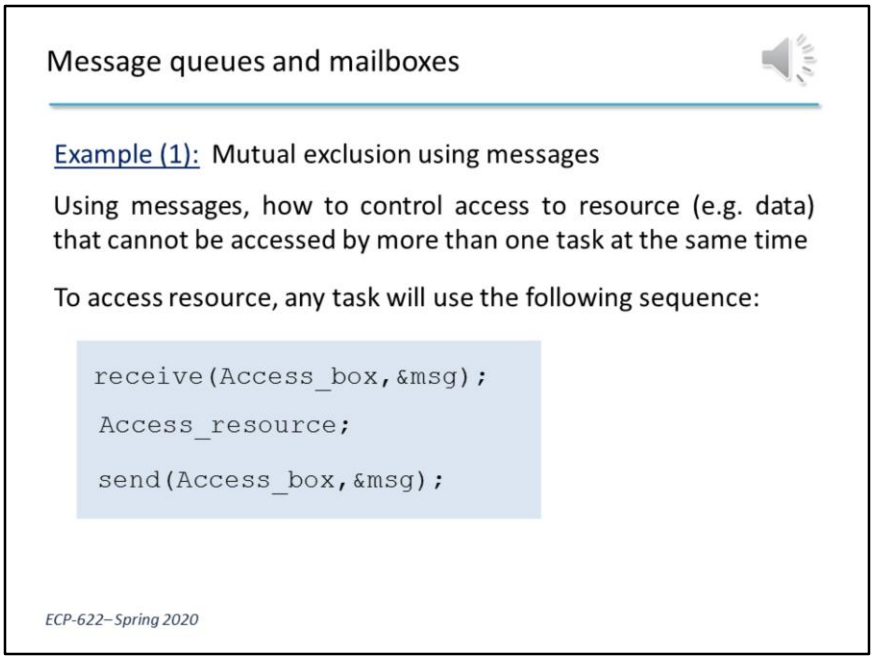
Alternatively, task may attempt to receive a message that was not yet sent. Two options exist: it can be blocked waiting for the message to arrive, or it may immediately return an error.

Message queues and mailboxes

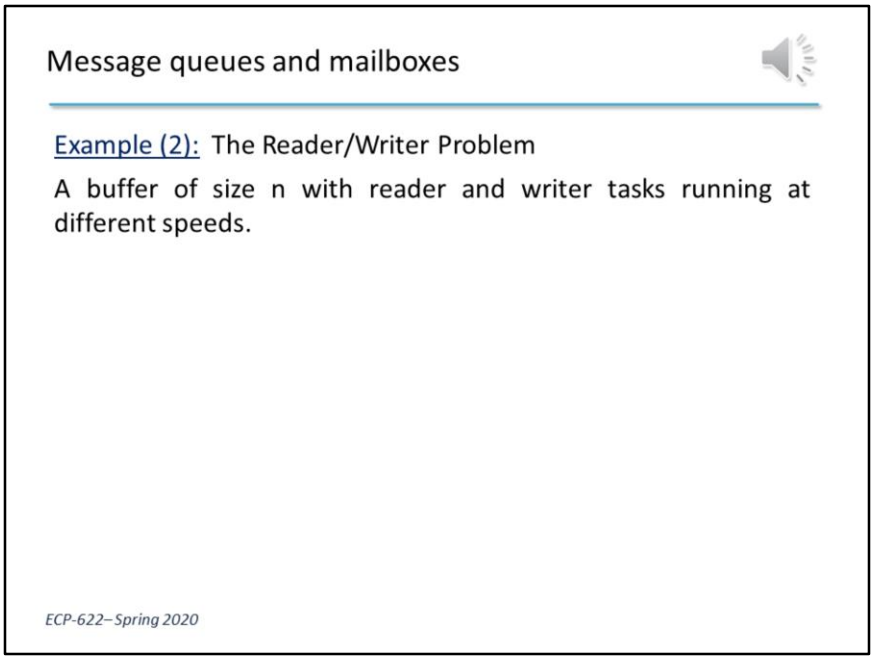
Example (1): Mutual exclusion using messages

Using messages, how to control access to resource (e.g. data) that cannot be accessed by more than one task at the same time

As an example, we consider again the problem of mutual exclusion solved before using semaphores.



Since we have communication among many tasks, we need a mailbox. To access resource, task waits to receive a message from the mailbox. After ending access, it returns the message to the mailbox. To work correctly, mailbox should be initialized with just one message placed in it.



As another example we consider again the problem of reader and writer with a bounded buffer.

Message queues and mailboxes

Example (2): The Reader/Writer Problem

A buffer of size n with reader and writer tasks running at different speeds.

Reader task

```
receive(writer,'full');
Read_data_item;
send(writer,'empty');
```


Before reading, reader waits for a message from writer indicating that one place in buffer was filled. After removing an item it sends a message to writer indicating that a place is emptied. Note that here we use direct addressing since communication is one-to-one.

Message queues and mailboxes

Example (2): The Reader/Writer Problem

A buffer of size n with reader and writer tasks running at different speeds.

Reader task

```
receive(writer,'full');
Read_data_item;
send(writer,'empty');
```

Writer task

```
receive(reader,'empty');
Write_data_item;
send(reader,'full');
```


As for the writer, it waits for a message indicating that a place is empty before writing. After writing it notifies the reader by a message. To work correctly we must initialize the operation by sending n messages with code 'empty' to the writer (assuming buffer is initially empty). As an exercise, consider how this will be modified with indirect addressing in case of many writers/many readers.

Queuing and Synchronization

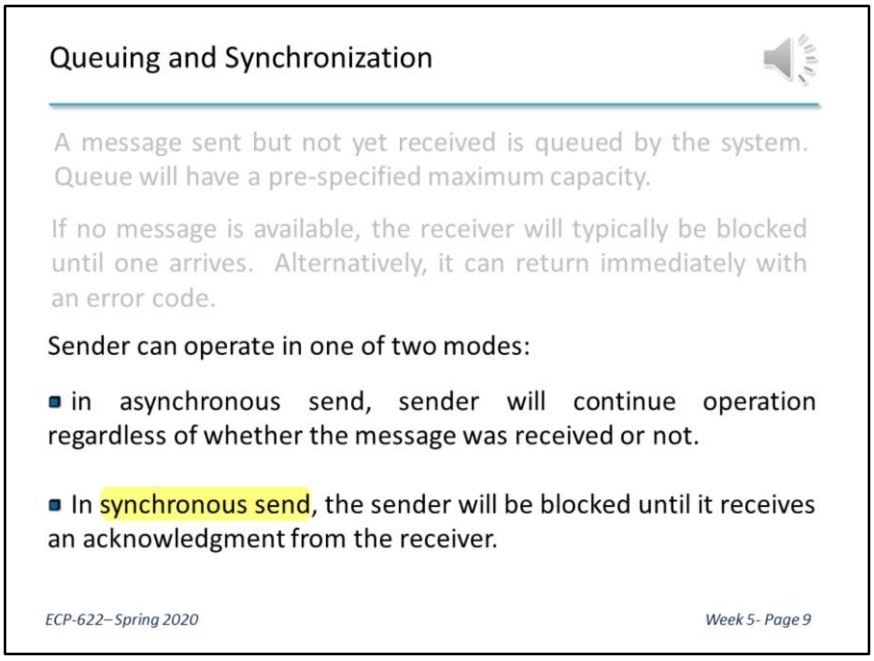
A message sent but not yet received is queued by the system. Queue will have a pre-specified maximum capacity.

If no message is available, the receiver will typically be blocked until one arrives. Alternatively, it can return immediately with an error code.

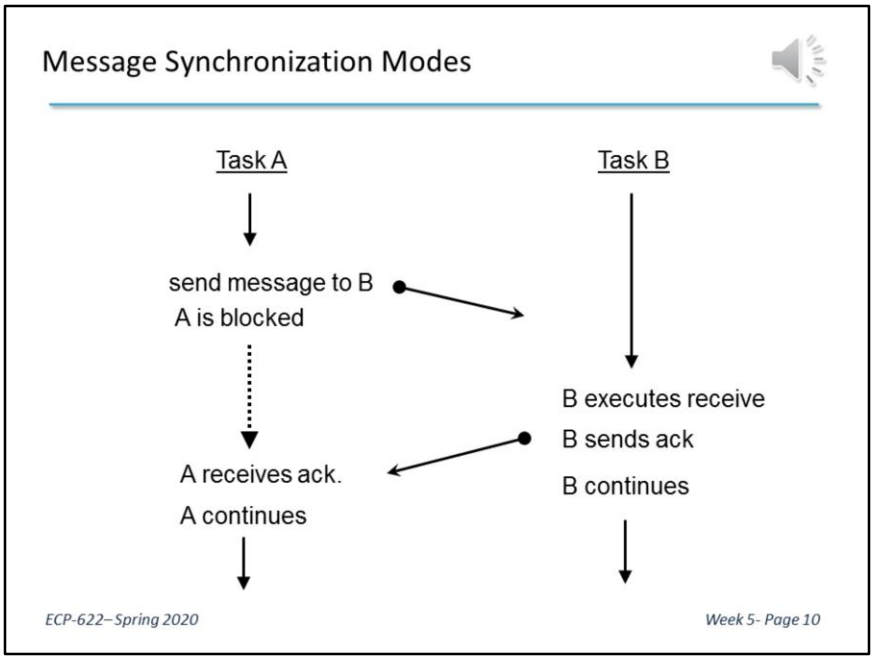
Sender can operate in one of two modes:

- in asynchronous send, sender will continue operation regardless of whether the message was received or not.

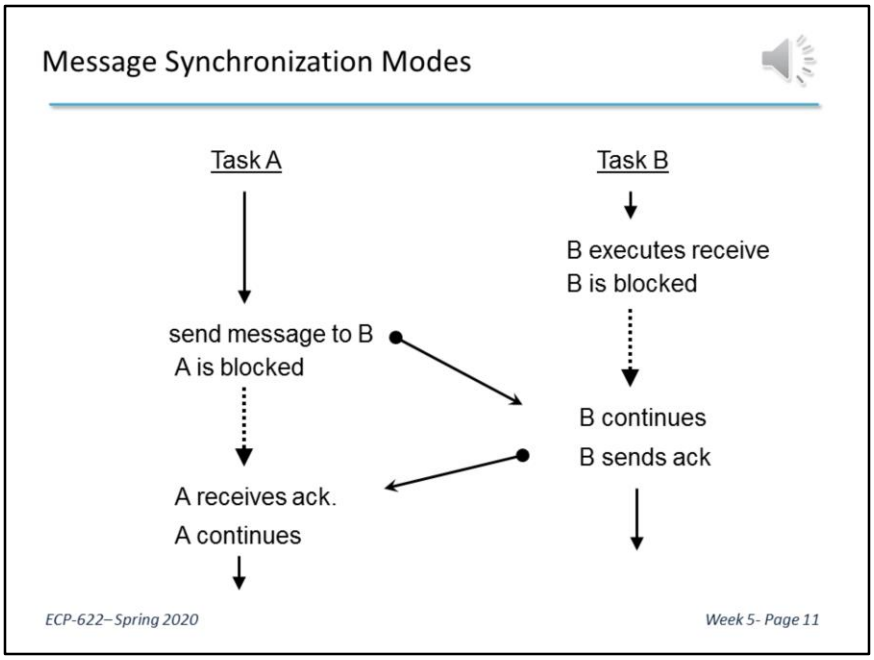
The receiver may be blocked, what about the sender? In asynchronous send option, sender just sends the message and do not wait for any thing.



In case messages may be lost (e.g. if sent over a network) it may be necessary for the sender to be blocked waiting for an acknowledgment from the receiver. In this synchronous send mode, ideas similar to those used in network data link layer can be used. For example what if message arrives but the acknowledgment itself was lost?



In this diagram, vertical arrows indicate time axis. Here task A sent a message in synchronous mode. Task B received the message after a while and sent the acknowledgment that unblocks A when received.

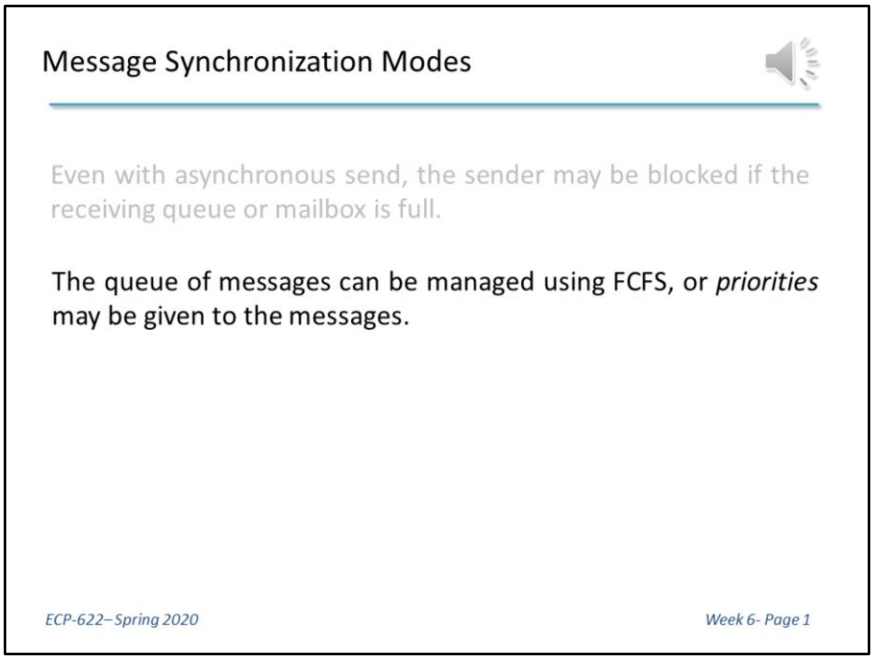


Here, B attempted to receive before A sent the message and hence is blocked. When message from A arrives, B receives it immediately and acknowledges it.

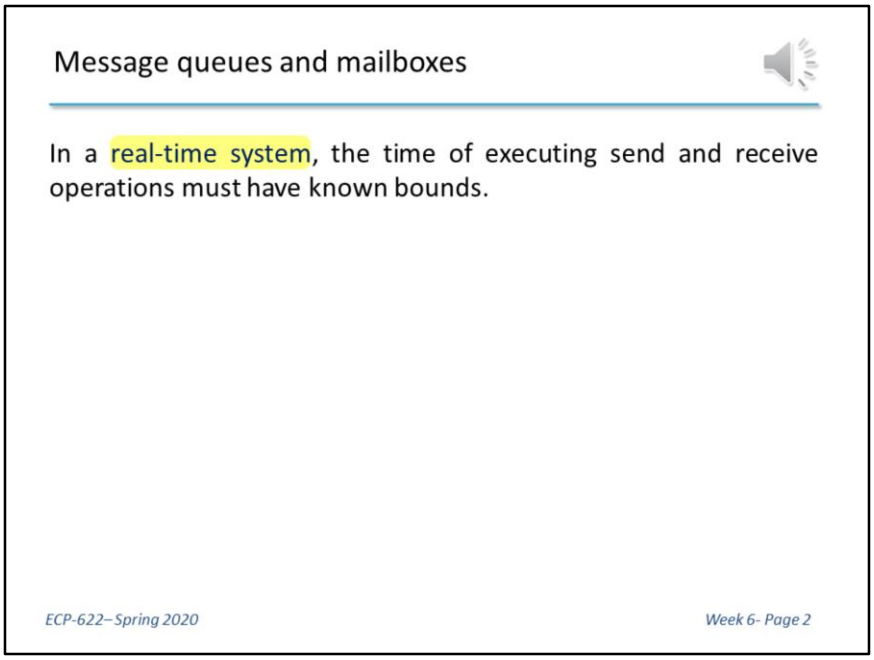
Message Synchronization Modes

Even with asynchronous send, the sender may be blocked if the receiving queue or mailbox is full.

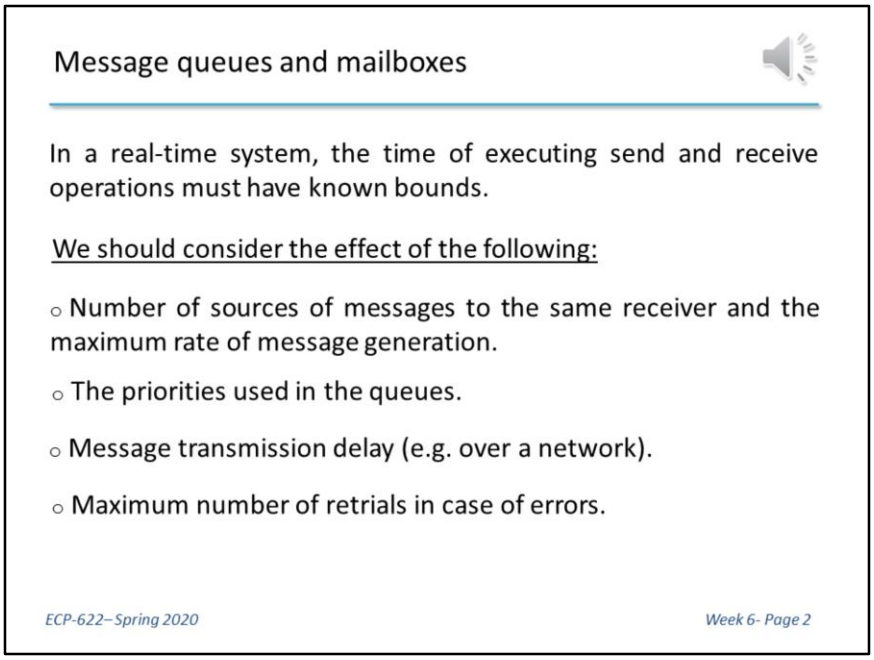
What if sender tries to send to a full queue of messages? It either blocks waiting for a place or immediately returns an error.



What if a task executes a receive operation with several pending messages. It may receive the first one sent, or else it receives the one with highest priority. Messages thus can have priority level normally related to the priority of tasks exchanging them.



All the above applies to all types of operating systems. In RTOS, the difference is that the time of send and receive operations are critical to have a predictable worst-case execution time for tasks.



For example, the delay of receiving a message will be a queuing delay with a maximum value depending on how many messages may be pending in the queue. Also, in case of synchronous send, maximum delay occurs if an acknowledgment is not received and send is reattempted for several times.