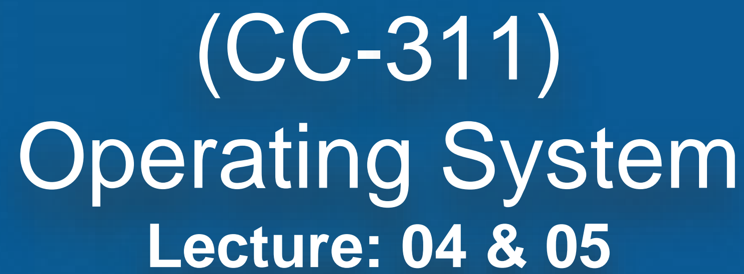


Professor: Syed Mustaghees Abbas

Fair-share scheduling

- Fair-share scheduling is a scheduling strategy for computer operating systems in which the CPU usage is equally distributed among system users/groups, as opposed to equal distribution among processes.
- For example, if four users (A,B,C,D) are concurrently executing one process each, the scheduler will logically divide the available CPU cycles such that each user gets 25% of the whole (100% / 4 = 25%).
- If user B starts a second process, each user will still receive 25% of the total cycles, but each of user B's processes will now use 12.5%. On the other hand, if a new user starts a process on the system, the scheduler will reapportion the available CPU cycles such that each user gets 20% of the whole (100% / 5 = 20%).

Fair-share scheduling

- We can also partition users into groups, and apply the fair share algorithm to the groups as well.
- Now the available CPU cycles are divided first among the groups, then among the users within the groups, and then among the processes for that user.
- For example, if there are three groups (1,2,3) containing three, two, and four users respectively, the available CPU cycles will be distributed as follows:
  - ✓ 100% / 3 groups = 33.3% per group
  - ✓ Group 1: (33.3% / 3 users) = 11.1% per user
  - ✓ Group 2: (33.3% / 2 users) = 16.7% per user
  - ✓ Group 3: (33.3% / 4 users) = 8.3% per user

Fair-share scheduling

- One common method of logically implementing the fair-share scheduling strategy is to recursively apply the round-robin scheduling strategy at each level of abstraction (processes, users, groups, etc.)
- The time quantum required by round-robin is arbitrary, as any equal division of time will produce the same results.

Priority inversion

- In computer science, priority inversion is a problematic scenario in scheduling in which a high priority task is indirectly preempted by a medium priority task effectively "inverting" the relative priorities of the two tasks.
- As an example, assume we have three processes, L, M, and H, whose priorities follow the order L < M < H.
- Assume that process H requires resource R, which is currently being accessed by process L.
- Ordinarily, process H would wait for L to finish using resource R. However, now suppose that process M becomes runnable, thereby preempting process L.
- Indirectly, a process with a lower priority—process M has affected how long process H must wait for L to relinquish resource R.

Priority inversion (Continued)

- This problem, known as priority inversion, can be solved by use of the **priority-inheritance protocol**.
- According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question.
- When they are finished, their priorities revert to their original values.
- In the example above, a priority-inheritance protocol allows process L to temporarily inherit the priority of process H, thereby preventing process M from preempting its execution.
- When process L has finished using resource R, it relinquishes its inherited priority from H and assumes its original priority. As resource R is now available, process H not M will run next.

LOTTERY SCHEDULING

- Gives variable numbers of lottery tickets to processes
- Holds lotteries to decide which thread will get the CPU

Traditional schedulers

- Priority schemes:
  - ✓ Priority assignments often ad hoc : highest priority always wins
  - ✓ Priority inversion: high-priority jobs can be blocked behind low-priority jobs
- “Fair share” schemes adjust priorities with a feedback loop to achieve fairness
  - ✓ Only achieve long-term fairness

LOTTERY SCHEDULING (Continued)

- Priority determined by the number of tickets each thread has:
  - ✓ Priority is the relative percentage of all of the tickets whose owners compete for the resource
- Scheduler picks winning ticket randomly, gives owner the resource

Example

- Three threads
  - ✓ A has 5 tickets
  - ✓ B has 3 tickets
  - ✓ C has 2 tickets
- If all compete for the resource
  - ✓ B has 30% chance of being selected
- If only B and C compete
  - ✓ B has 60% chance of being selected

LOTTERY SCHEDULING (Continued)

- Lottery scheduling is **starvation-free**
  - ✓ Every ticket holder will finally get the resource
- Lottery scheduling is **probabilistically fair**
  - ✓ If a thread has a t tickets out of T
  - ✓ Its probability of winning a lottery is $p = t/T$

Transfers of tickets

- Explicit transfers of tickets from one client to another
- They an be used whenever a client blocks due to some dependency
  - ✓ When a client waits for a reply from a server, it can temporarily transfer its tickets to the server
- They eliminate **priority inversions**

LOTTERY SCHEDULING (Continued)

Ticket Inflation

- Lets users create new tickets
  - Like printing their own money
- **Normally disallowed** except among mutually trusting clients
  - Lets them adjust their priorities dynamically without explicit communication

Ticket currencies

- Consider the case of a user managing multiple threads
  - Want to let her favor some threads over others
  - Without impacting the threads of other users
- Will let her create new tickets but will debase the individual values of all the tickets she owns
  - Her tickets will be expressed in a **new currency** that will have a variable **exchange rate** with the **base currency**

LOTTERY SCHEDULING (Continued)

- Ann manages three threads
  - ✓ A has 5 tickets
  - ✓ B has 3 tickets
  - ✓ C has 2 tickets
- Ann creates 5 extra tickets and assigns them to process C
  - ✓ Ann now has 15 tickets
- These 15 tickets represent 15 units of a new currency whose exchange rate with the base currency is 10/15
- The total value of Ann tickets expressed in the base currency is still equal to 10

Cooperating Processes

- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process

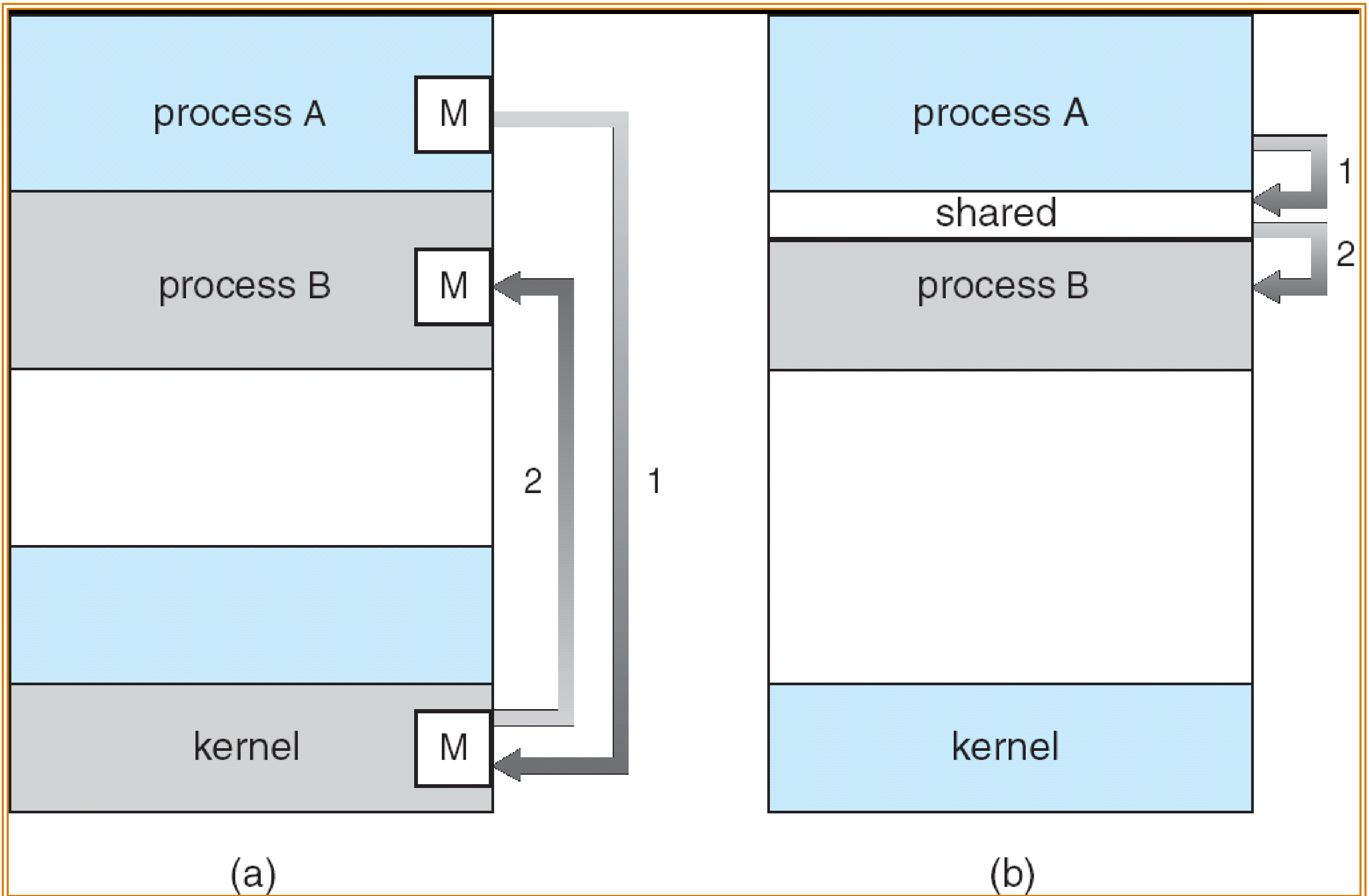
Need for Cooperating Processes

- There are several reasons for providing an environment that allows process cooperation:
  - ✓ **Information sharing**. Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
  - ✓ **Computation speedup**. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).
  - ✓ **Modularity.** We may want to construct the system in a modular fashion, dividing the system functions into separate processes.
  - ✓ **Convenience**. Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

Models of IPC

- There are two fundamental models of inter-process communication:
  - ✓ **Shared Memory**
  - ✓ **Message Passing**
- In the **shared-memory model**, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.
- In the **message passing model**, communication takes place by means of messages exchanged between the cooperating processes.

Models of IPC (Continued)



Models of IPC (Continued)

- **Explanation of the Diagram of Models of IPC**

  In the above diagram
  - ✓ "(a)" represents message passing.
  - ✓ "(b)" represents shared memory.
- **Shared memory (b):**
  - ✓ Processes A and B each have their own private memory space.
  - ✓ They communicate by reading and writing to a shared memory region, which is accessible to both processes.
  - ✓ The kernel manages access to the shared memory region to ensure data consistency.
- **Message passing (a):**
  - ✓ Processes A and B communicate by exchanging messages through the kernel.
  - ✓ Each process sends messages to the kernel, specifying the receiver process and the data to be sent.
  - ✓ The kernel is responsible for delivering the messages to the specified receiver process.

Models of IPC (Continued)

Both of the models just discussed are common in operating systems.

- Message passing is useful for exchanging smaller amounts of data.
- Shared memory allows maximum speed and convenience of communication, as it can be done at memory speeds when within a computer and requires no kernel intervention.

Shared Memory Systems

Producer-consumer problem

- To illustrate the concept of cooperating processes, let's consider the Producer-consumer problem, which is a common paradigm for cooperating processes.
- A producer process produces information that is consumed by a consumer process. For example a web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource.
- One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can **be filled by the producer and emptied by the consumer**. This buffer will reside in a region of memory that is shared by the producer and consumer processes.

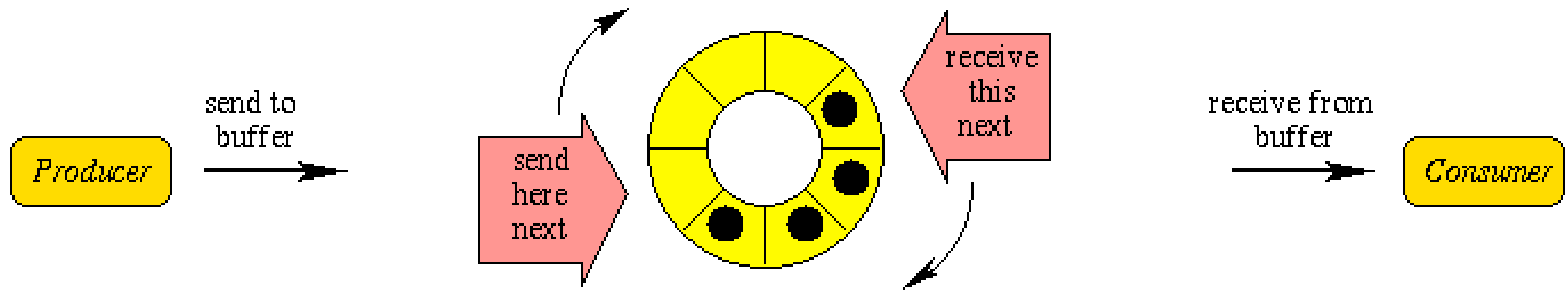
Producer-consumer problem (Continued)

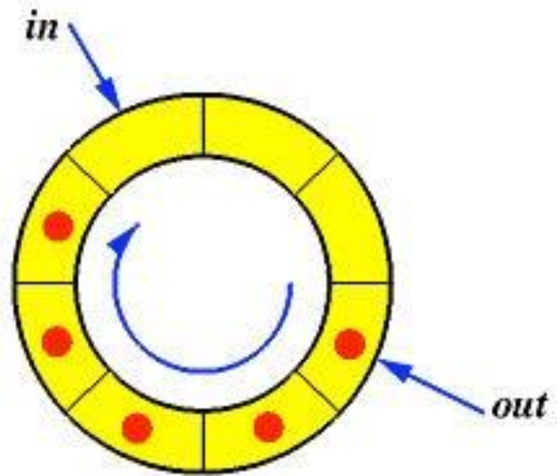
Now we look at how the bounded buffer can be used to enable processes to share memory. The following variables reside in a region of memory shared by the producer and consumer processes:

```
#define BUFFER_SIZE 10
int buffer [BUFFER_SIZE] ;
int in = 0 ;
int out = 0 ;
```

The shared buffer is implemented as a circular array with two logical pointers: in and out. The variable **in** points to the next free position in the buffer; **out** points to the first full position in the buffer. The buffer is empty when in == out; the buffer is full when
((in + 1) % BUFFER_SIZE) == out.

Producer-consumer problem (Continued)





Producer-consumer problem (Continued)

```
while (true) {
    /* Produce an item */
    while (((in + 1) % BUFFER_SIZE) == out)
        ;  //no free buffers
        buffer[in] = item;
       in = (in + 1) % BUFFER_SIZE;
}
```

Producer

```
while (true) {
     while (in == out)
         ; //nothing to consume

     //remove an item from the buffer
     item = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     return item;
}
```

Consumer

Producer-consumer problem (Continued)

- How many items can be stored in a buffer of size in previous example?

Producer-consumer problem (Continued)

```
#define BUFFER_SIZE 10
int buffer [BUFFER_SIZE] ;
int in = 0 ,
int out = 0 ;
int count = 0;
```

Producer-consumer problem (Continued)

- How many items can be stored in a buffer of size in previous example?
  - ✓ Our solution allowed at most BUFFER.SIZE - 1 items in the buffer at the same time (refer to old slides).
  - ✓ To remedy this deficiency we can add an integer variable count, initialized to 0.
  - ✓ Count is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.

Producer-consumer problem (Continued)

```
while (true) {
    /* Produce an item */
    while (count == BUFFER SIZE)
        ; //no free buffers
        buffer[in] = item;
        in = (in + 1) % BUFFER SIZE;
        count = count+1;
}
```

Producer

```
while (true) {
    while (count == 0)
        ; //nothing to consume
    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    count = count-1;
    return item;
}
```

Consumer

Message Passing Systems

- Message passing provides a mechanism to allow processes to communicate without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
- For example, a chat program used on the World Wide Web could be designed so that chat participants communicate with one another by exchanging messages.
- A message-passing facility provides at least two operations: **send(message)** and **receive(message)**. If processes P and Q want to communicate, they must send messages to and receive messages from each other; a communication link must exist between them.

Message Passing Systems (Continued)

- We look at issues related to the implementation of message passing systems:

1. Naming:

- In message passing communication can be either direct or indirect.
- In **direct communication**, each process must explicitly (clearly) name the recipient or sender of the communication.
- *send(P, message)*—Send a message to process P.
- *receive(Q, message)*—Receive a message from process Q.
- The processes need to know only each other’s identity to communicate.

Message Passing Systems (Continued)

- A link is associated with exactly two processes.
- This scheme exhibits **symmetry** in addressing; that is, both the sender process and the receiver process must name the other to communicate.
- A variant of this scheme employs **asymmetry** in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender.
- *send(P, message)*—Send a message to process P.
- *receive(id, message)*—Receive a message from any process
- Both approaches have disadvantage that changing the identifier of a process may necessitate updation of all references to the old identifier.

Message Passing Systems (Continued)

- With **indirect communication**, the messages are not sent directly from sender to receiver but to a shared data structure consisting of queues (mailbox) that can temporarily hold.

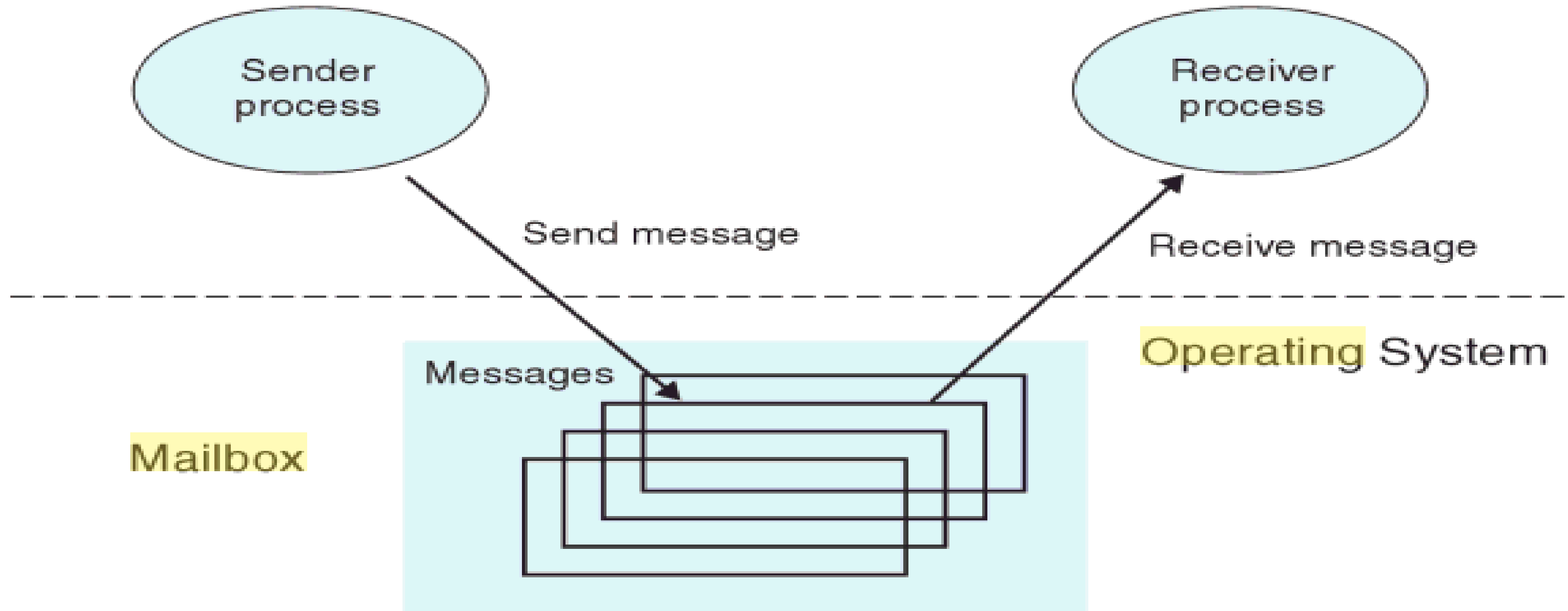


Figure 6.10 Two processes communicating via a mailbox.

Message Passing Systems (Continued)

- While communication, one process sends message to a mailbox and other process receives it from mailbox.
- Each **mailbox** has a **unique identification**. For example **pipes in Linux**
- *send(A, message)*—Send a message to mailbox A.
- *receive(A, message)*—Receive a message from mailbox A.

Message Passing Systems (Continued)

2. Synchronization:

- Message passing may be either blocking or non-blocking— also known as synchronous and asynchronous.
- **Blocking send**: The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Non-blocking send**: The sending process sends the message and resumes operation.
- **Blocking receive:** The receiver blocks until a message is available.
- **Non-blocking receive**: The receiver retrieves either a valid message or continue operation.

Message Passing Systems (Continued)

3. Buffering:

- Direct or indirect, messages exchanged by communicating processes reside in a temporary queue which can be implemented in three ways:
- **Zero capacity:** The queue has a maximum length of zero. In this case, the sender must block until the recipient receives the message.
- **Bounded capacity:** The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue and the sender can continue execution without waiting, however. If the link is full, the sender must block until space is available in the queue.
- **Unbounded capacity**: The queues length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.