CISC 7310X
C03b: Inter-Process Communication

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College

Acknowledgement

- This slides are a revision of the slides by the authors of the textbook

Outline

- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***

Cooperating Processes

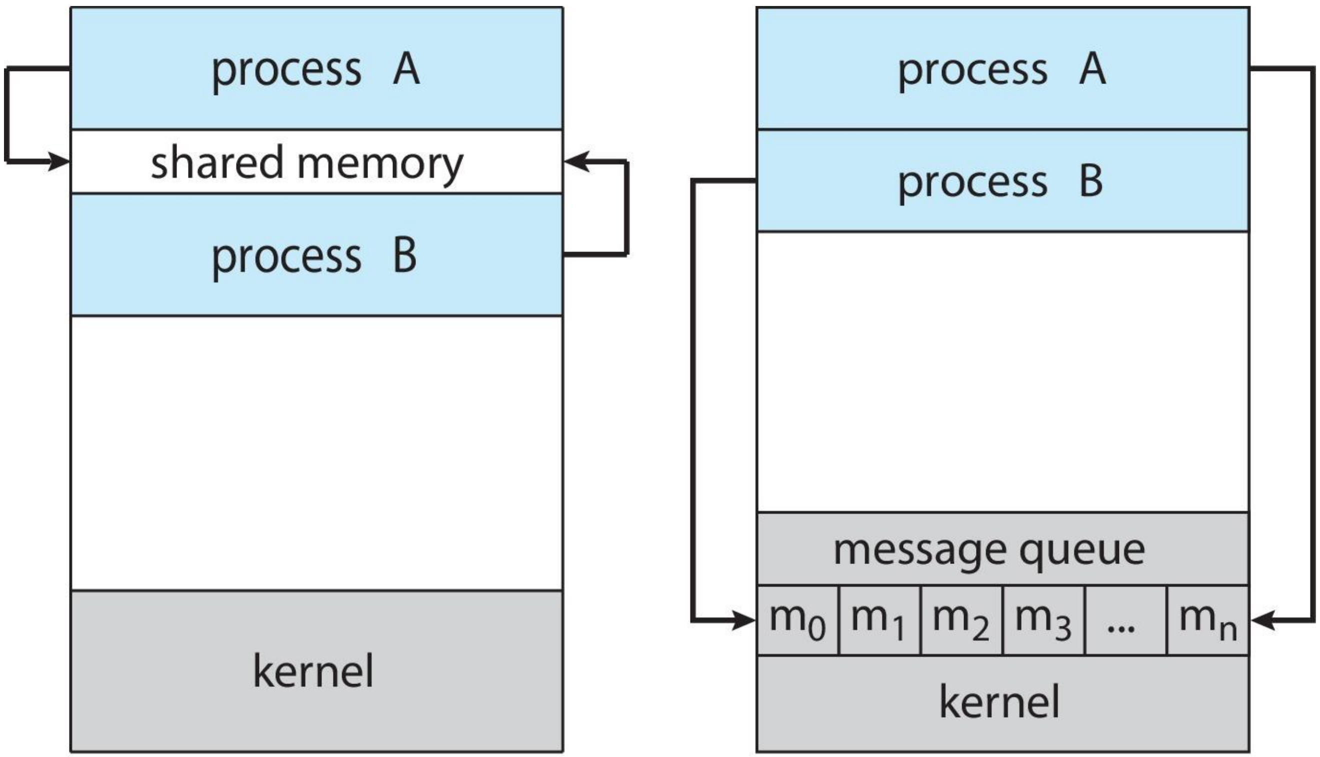
- ***Independent*** process cannot affect or be affected by the execution of another process
- ***Cooperating*** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Interprocess Communication

- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

(a) Shared memory. (b) Message passing.



(a) (b)

Questions?

- Concept and benefits of interprocess communciation

Producer-Consumer Problem

- Paradigm for cooperating processes
- *Producer* process produces information that is consumed by a *consumer* process
- The information is stored in a memory buffer
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to **synchronize** their actions when they access shared memory.
- Synchronization is discussed in great details in a few weeks

Bounded-Buffer: Shared-Memory Solution

- Shared data
- Producer
- Consumer
- At present, we do not address concurrent access to the shared memory by the producer and the consumer.

Shared Data via Shared Memory

- Share BUFFER_SIZE – 1 items

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Producer Process via Shared Memory

```
item next_produced;

while (true) {
  /* produce an item in next produced */
  while (((in + 1) % BUFFER_SIZE) == out)
      ; /* do nothing */
  buffer[in] = next_produced;
  in = (in + 1) % BUFFER_SIZE;
}
```

Consumer Process via Shared Memory

```
item next_consumed;

while (true) {
      while (in == out)
            ; /* do nothing */
      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;

      /* consume the item in next consumed */
}
```

Questions?

- Producer-consumer problem
- Shared memory

Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*)
  - **receive**(*message*)
- The *message* size is either fixed or variable

Message Passing: Implementation Issues

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Communication Link

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication: Operations and Primitives

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

  send(*A*, *message*) – send a message to mailbox *A*

  receive(*A*, *message*) – receive a message from mailbox *A*

Indirect Communication: Mailbox Sharing?

- Mailbox sharing
    - P_1, P_2, and P_3 share mailbox *A*
    - P_1, sends; P_2 and P_3 receive
    - Who gets the message?
- Solutions
    - Allow a link to be associated with at most two processes
    - Allow only one process at a time to execute a receive operation
    - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Questions?

- Concept of message passing
- Implementation issues
- Direction and indirect communications

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

Producer-Consumer via Message Passing

- Producer
- Consumer

Producer via Message Passing

```
message next_produced;

while (true) {
    /* produce an item in
 next_produced */

        send(next_produced);

}
```

Consumer via Message Passing

```
message next_consumed;

while (true) {
     receive(next_consumed)

     /* consume the item in
 next_consumed */

}
```

Buffering

- Queue of messages attached to the link.
- Implemented in one of three ways
  1. Zero capacity – no messages are queued on a link. Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages Sender must wait if link full
  3. Unbounded capacity – infinite length Sender never waits

Questions?

- Producer-consumer problem via message passing
- Buffering for message passing

Examples of IPC System

- POSIX

POSIX Shared Memory

- Process first creates shared memory segment
  `shm_fd = shm_open(name, O CREAT | O RDWR, 0666);`
- Also used to open an existing segment
- Set the size of the object

  `ftruncate(shm_fd, 4096);`
- Use **`mmap()`** to memory-map a file pointer to the shared memory object
- Reading and writing to shared memory is done by using the pointer returned by **`mmap()`**.

Mach Message Passing

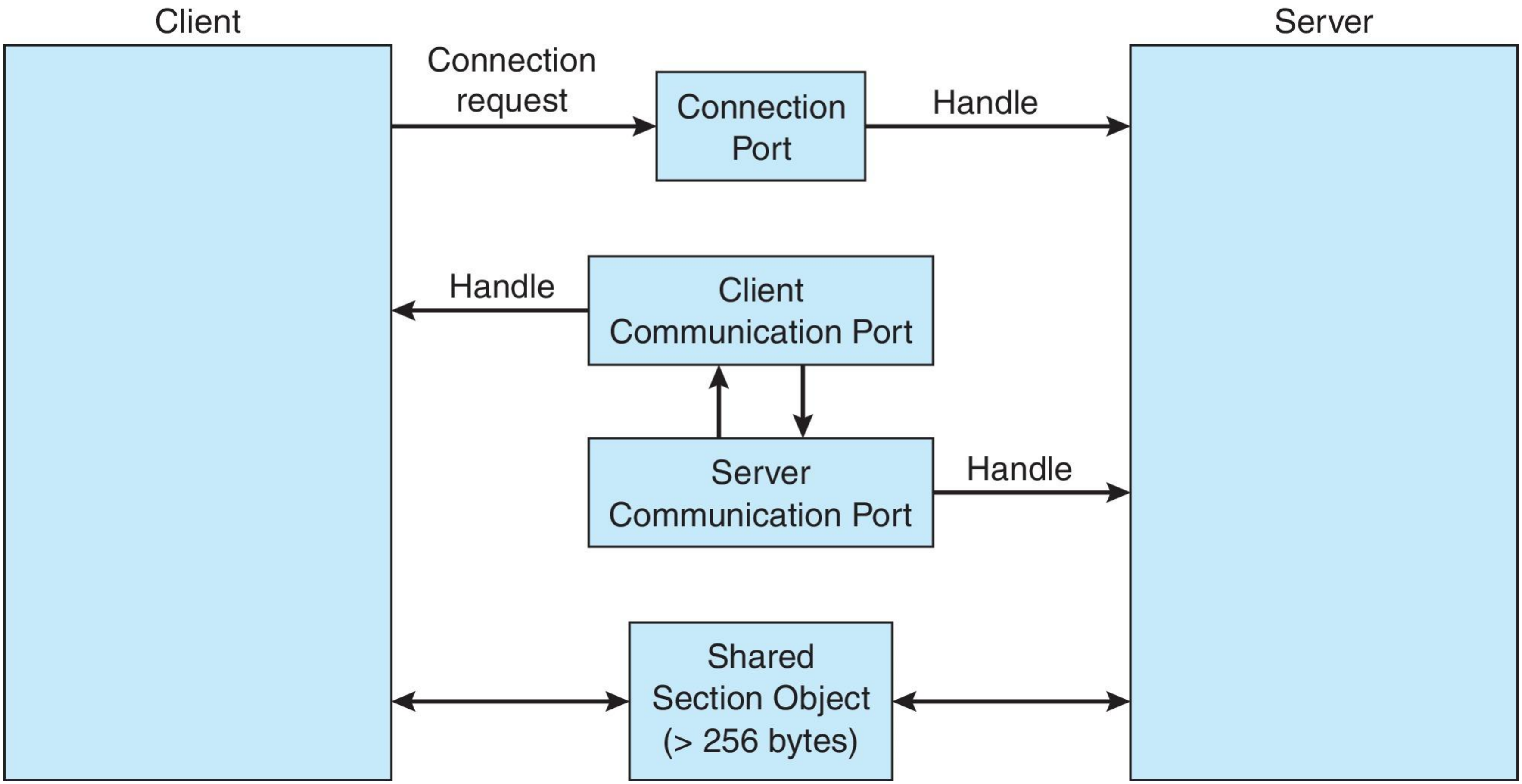
- Mach communication is message based
  - Even system calls are messages
  - Each task gets two ports at creation- Kernel and Notify
  - Messages are sent and received using the **`mach_msg()`** function
  - Ports needed for communication, created via

    `mach_port_allocate()`
  - Send and receive are flexible, for example four options if mailbox full:
    - Wait indefinitely
    - Wait at most n milliseconds
    - Return immediately
    - Temporarily cache a message

Windows IPC

- Message-passing centric via **advanced local procedure call** (**LPC**) facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - The client opens a handle to the subsystem's **connection port** object.
    - The client sends a connection request.
    - The server creates two private **communication ports** and returns the handle to one of them to the client.
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Local Procedure Call (LPC)



Questions?

- IPC in POSX
- IPC in Mach
- IPC in Windows

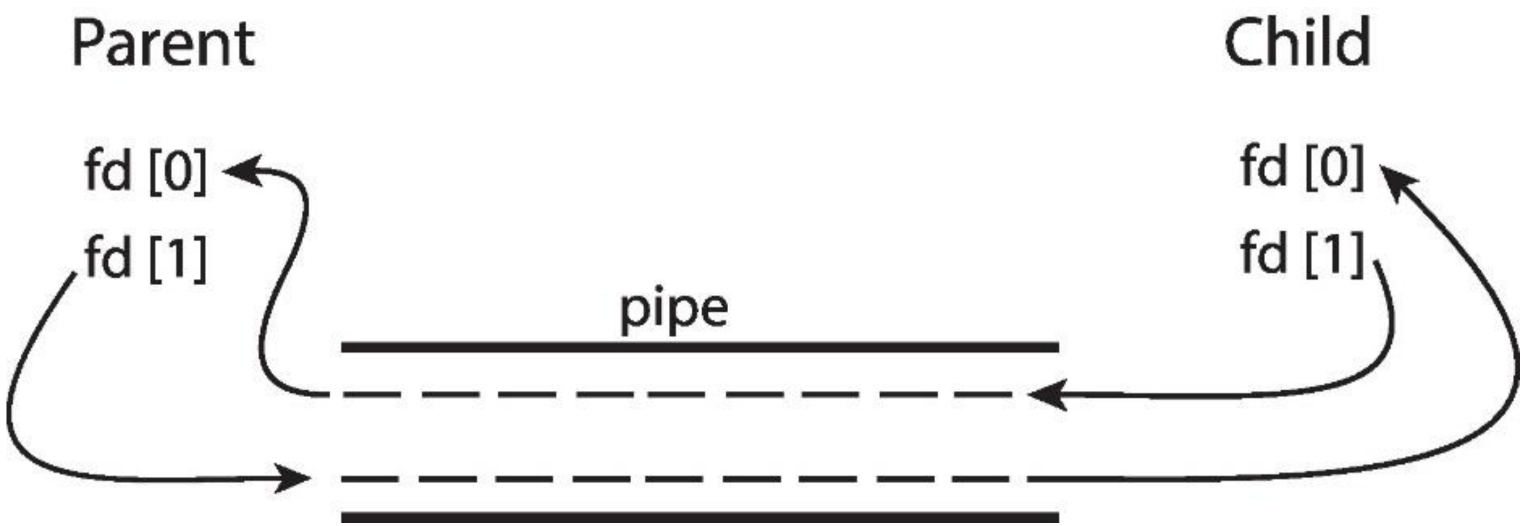
Pipes

- Acts as a conduit allowing two processes to communicate
  - The communication pattern follows message passing, but may be implemented using shared memory
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?
- **Ordinary pipes** – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- **Named pipes** – can be accessed without a parent-child relationship

Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style: unidirectional
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes
- Windows calls these **anonymous pipes**

Ordinary Pipes: Parent-Child relationship



Named Pipes

- Named Pipes are more powerful than ordinary pipes
  - Communication is bidirectional
  - No parent-child relationship is necessary between the communicating processes
  - Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

Questions?

- Concept of pipes
- Ordinary pipes
- Named pipes