Multiprocess Architecture – Chrome Browser

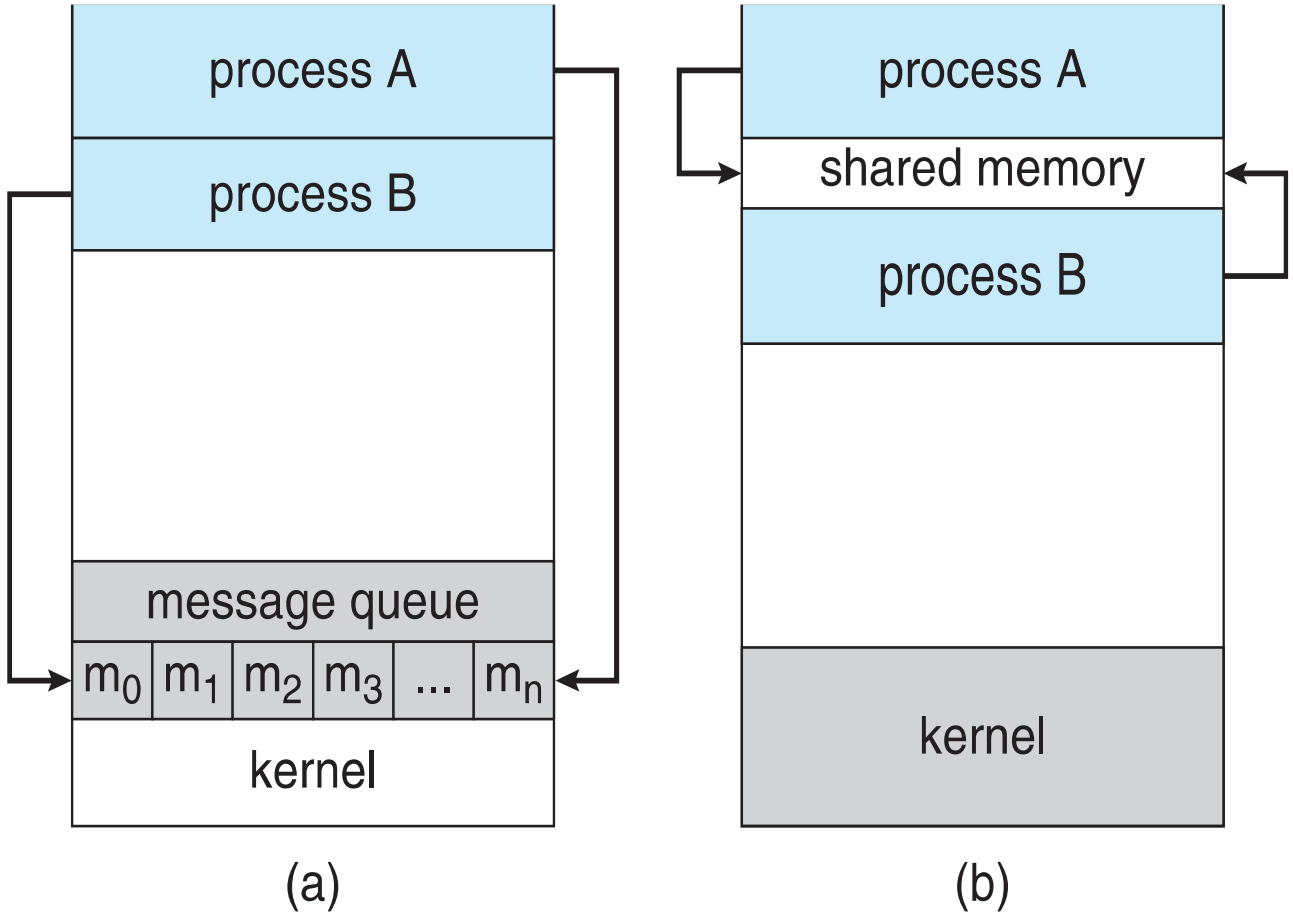
- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multi-process with 3 different types of (**communicating**) processes:
  - A **browser** process manages user interface, disk and network I/O
  - One or more **renderer** processes renders web pages, deals with HTML, Javascript, etc. A new renderer created for each tab/website opened
    - Since each tab is a separate process, they don't share memory. They also do not share file resources based on how their parent process (the browser) created them, thus minimizing effect of security exploits.
    - If a renderer crashes, it doesn't bring down the entire browser.
  - One or more **plug-in** processes for each type of plug-in

3.4 Inter-process Communication (IPC)

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing (e.g. shared file such as a database)
  - Computation speedup (if system has multiple CPU cores)
  - Modularity (may divide a program into tasks, may feed or use services of other tasks)
  - Convenience (e.g. a user may be editing while a spell check is running)
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

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



Producer-Consumer Problem

- The producer-consumer problem is a common paradigm for cooperating processes.
- The *producer* process produces information that is consumed by a *consumer* process, e.g.
  - Multiple subtasks forming wider function such as a compiler producing an object file and a linker task consuming object files to produce the executable code.
  - A client producing window commands (e.g. draw a rectangle) and an X11 display server consuming window commands.
  - An X11 display server producing mouse/keyboard data and a client process receiving mouse clicks or keyboard keys.
  - These were general examples of processes producing and consuming information.
    - X11 servers generally do not communicate using message passing (via Unix pipes or TCP/IP sockets).

3.4.1 Shared memory systems

- An area of **memory shared among the processes** that wish to communicate (the creation of this area is facilitated by the OS kernel since each process normally has a separate address space)
- After the shared memory has been created by the OS, **the mechanism used for communication** between the user processes is administered by them, not the operating system.
- A major issue is to provide a mechanism that allows the user processes to **synchronize** their actions when they access shared memory locations.
  - The communicating processes may use synchronization functions provided by the OS kernel. This shall be discussed in great details in the next chapter.

Producer-Consumer Problem

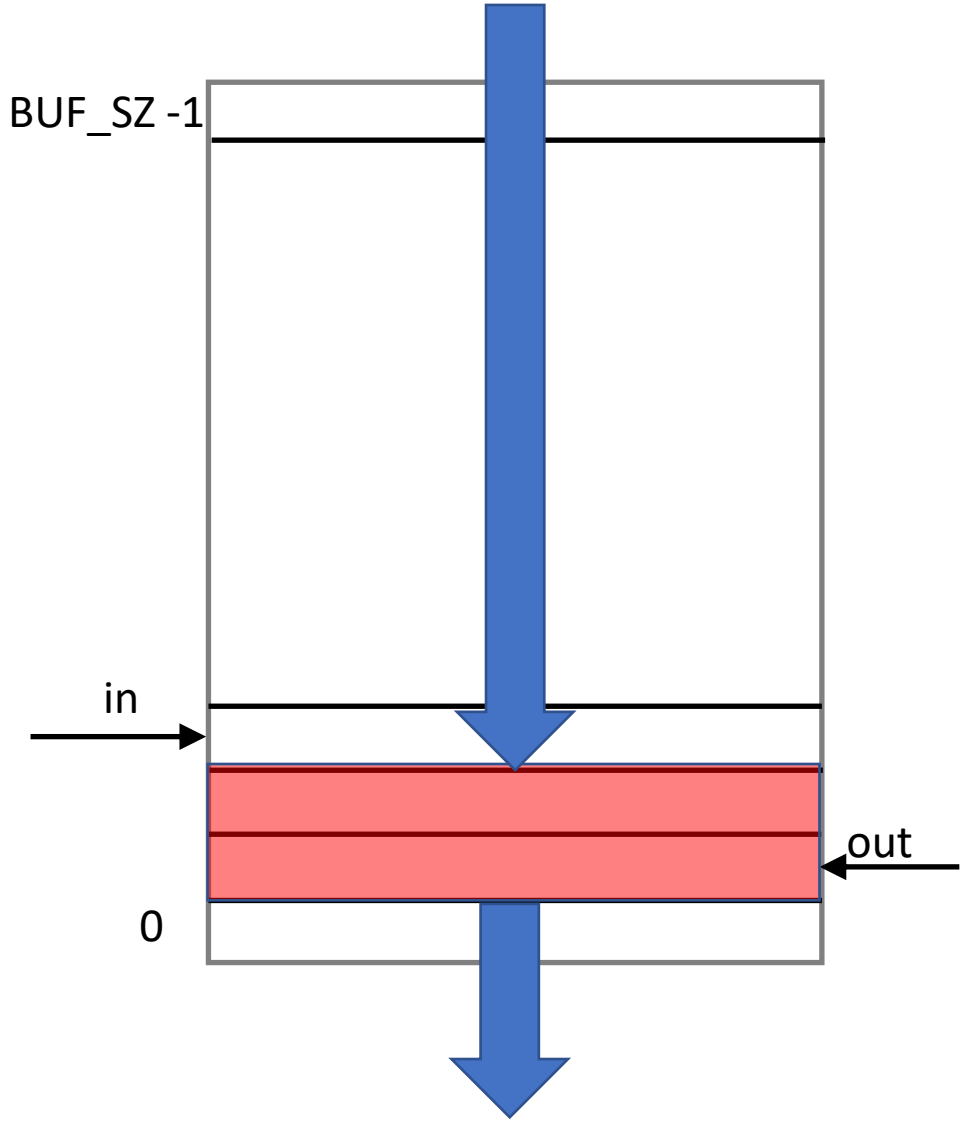
- The producer-consumer problem may illustrate issues with shared memory systems.
- Two types of buffers may be used
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data:

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

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

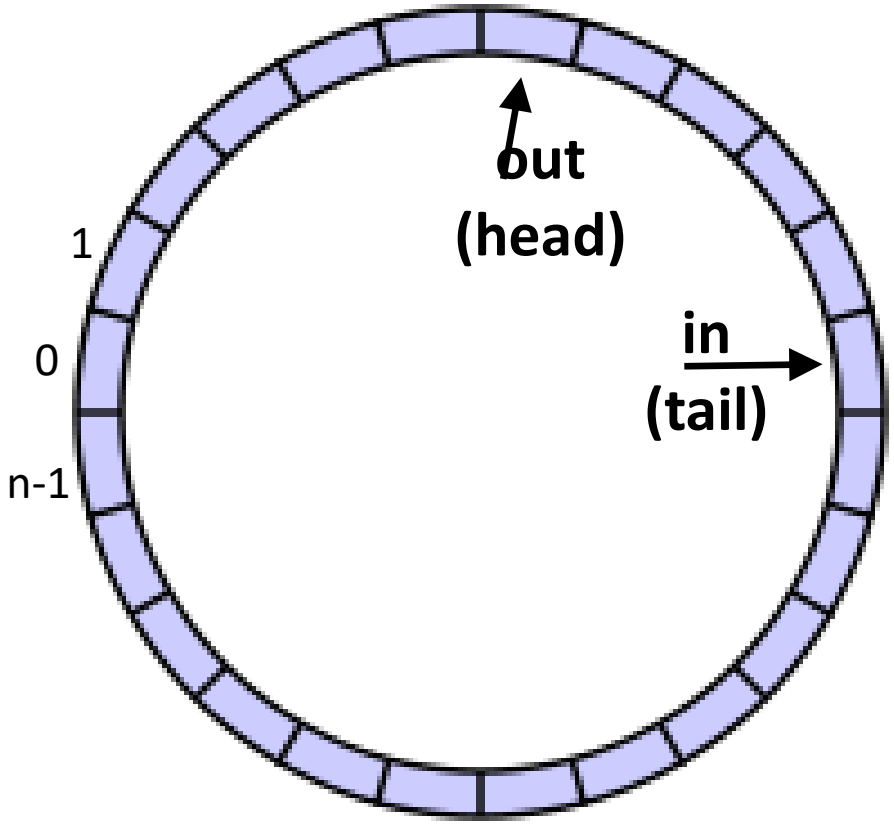
- Buffer needs to be administered and used as a FIFO or Queue



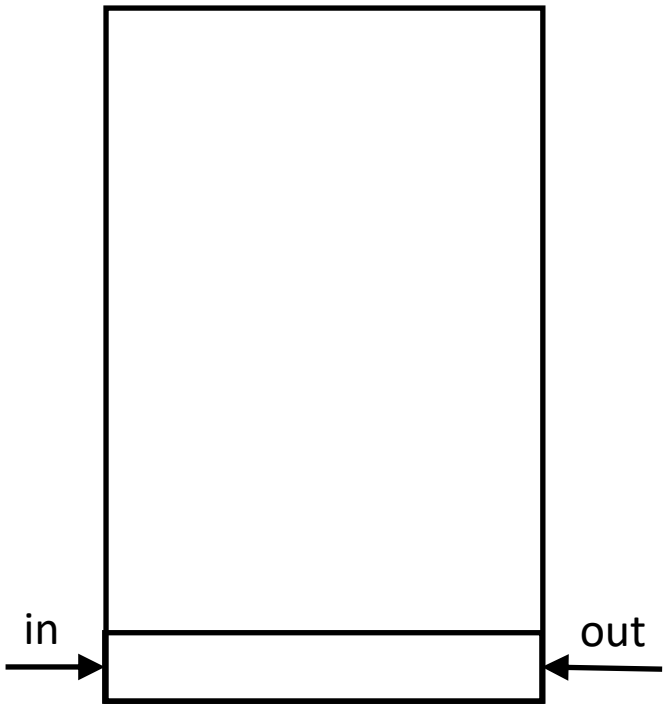
Bounded-Buffer – Shared-Memory Solution

- Shared data:

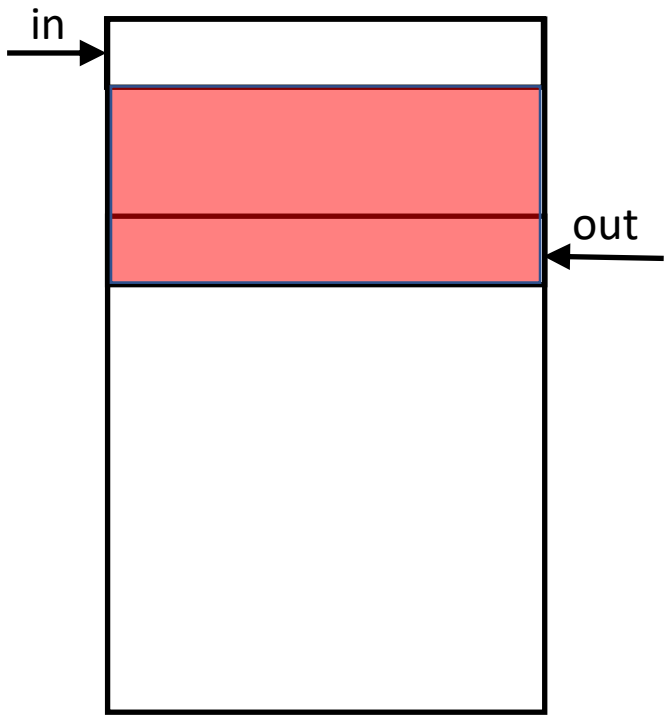
```
#define BUF_SZ 10
typedef struct {
  . . .
} item;

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



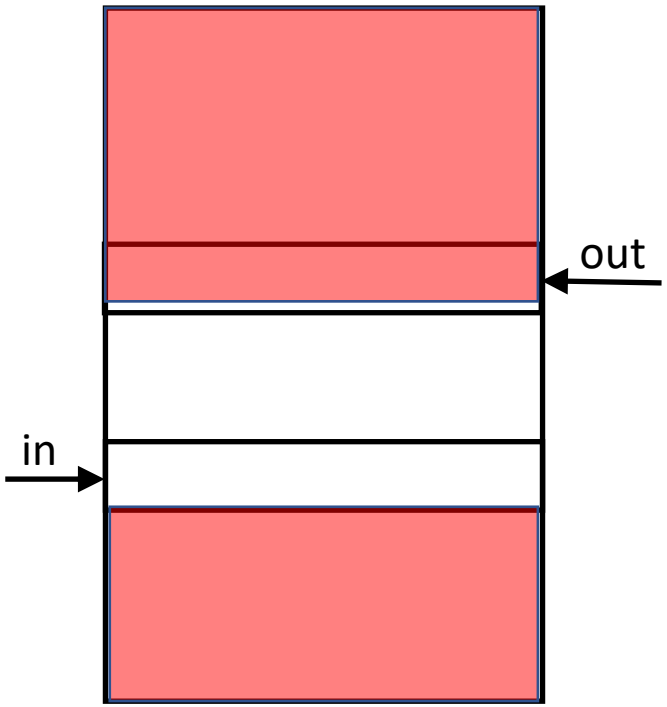
- Buffer needs to be administered and used as a FIFO or Queue



Initial state



After a few writes/reads - No wrapping OR Both ‘in” and “out” wrapped



After a few writes/reads - Only the “in” wrapped but not the “out” index

Bounded-Buffer – Producer

```
item next_produced;
while (true) {
  /* produce an item in next produced */

  /* wait till next in != out */
  while (((in + 1) % BUF_SZ) == out);

  buffer[in] = next_produced;
  in = (in + 1) % BUF_SZ;
}
```

Bounded Buffer – Consumer

```
item next_consumed;
while (true) {
  /* wait till in != out*/
  while (in == out);
  next_consumed = buffer[out];
  out = (out + 1) % BUF_SZ;

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

- Solution is correct, but can only use BUFFER_SIZE-1 elements (why?)
- In this solution, the producer and consumer do not access the same item simultaneously
- What happens if both need to access the same location concurrently (e.g. a shared variable or a shared counter?
  - Synchronization will then be needed (next chapter) → can't do that for now

3.5 Examples of IPC Systems – POSIX shared memory

- A process first creates shared memory segment using **shm_open**

```
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

- Processes needing to communicate via shared memory must know the name of this memory-mapped object. The last parameter is the file permissions. The function returns a file descriptor.
- Same function is also used to open an existing segment to share it
- A process may then use **ftruncate** to set the size of the object (in bytes).
- **mmap** may then be used to map and obtain a pointer to the shared memory.
- Now the process could write to the shared memory, e.g.

```
sprintf(shared_mem_ptr, "Writing to shared memory");
```

POSIX – cont.

- In Unix/Linux, the shared memory is abstracted as a file system located at /dev/shm
- shm (or shmfs) is also known as tmpfs. tmpfs means temporary file storage facility. It is intended to appear as a mounted file system, but one which uses memory instead of a persistent storage device.

- NOTE: It may be useful to install the man pages for POSIX

  sudo apt-get install manpages-dev // you probably have this already

  sudo apt-get install manpages-posix-dev

IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

IPC POSIX Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

3.4.2 Message passing systems

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

Message Passing (Cont.)

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

Message Passing (Cont.)

- Implementation of communication link
  - On the physical level:
    - Via shared memory
    - Hardware bus or a communication network.
  - On the logical level:
    A. Direct or indirect
    B. Synchronous or asynchronous
    C. Automatic or explicit buffering

A . Naming: Direct Communication

- Symmetric schemes: Processes must specify each other explicitly (via process identifier):
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*Q, message*) – receive a message from process Q
- Asymmetric schemes: Only sender names the recipient
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*&ID, message*) – receive a message from any process and when you receive a message indicate the name of the sender (in the variable *ID*).
- Properties of communication link
  - Links are established automatically
    - In other words, you don't need to establish the link, you just send a message.
  - Link association:
    - A link is associated with exactly one pair of communicating processes
    - Between each pair there exists exactly one link
  - The link is usually bi-directional
- Process identifiers may be names or integer numbers.
- Disadvantage is that a process identifier needs to be hardcoded, which thus requires recompilation if the identifiers change.

Naming: Indirect Communication

- Messages are directed to, and received from **ports or mailboxes**. Primitives are defined as:
  - **`send`**(*A, message*) – send a message to mailbox A
  - **`receive`**(*A, message*) – receive a message from mailbox A
- Properties of communication link
  - Each mailbox has a unique identifier. (name or number)
  - Link established only if processes share a common mailbox
  - Link association:
    - A link **may** be associated with many processes
    - Each pair of processes **may** share several communication links
  - Link may be unidirectional or bi-directional

Naming: Indirect Communication – cont.

- A port (or mailbox) may be **owned by a process**
  - The port/mailbox is attached to a process and implemented inside its address space.
  - If the process exits, the mailbox is destroyed.
  - Unidirectional:
    - Owner (server) can only receive messages via the mailbox
    - User (client) sends message to the mailbox.
- Alternatively, a port may be **owned by the operating systems**, and thus the OS may need to provide operations such as:
  - Create a new port/mailbox
  - Delete the port.
  - Send and receive messages through the port
  - Unix implements two such ports;
    - pipes (unidirectional) and
    - TCP/IP ports (bidirectional)

Naming: Indirect Communication – cont.

- 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.

B. Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is accepted by OS.
    - The peculiar case of no-buffering is to be address later (zero buffering)
  - **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 and the link has zero buffering, we have a **rendezvous**
- NOTE: Throughout the course:

BLOCKING = SYNCHRONOUS

NON-BLOCKING = ASYNCHRONOUS

Synchronization (Cont.)

- Producer-consumer becomes trivial (i.e. no concern about how to manage a circular buffer

```
message next_produced;
while (true) {
  /* produce an item into
  next produced */
  send(next_produced);
}
```

```
message next_consumed;
while (true) {
 /* consume the item into
  next consumed */
  receive(next_consumed);
}
```

C. 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) to accept the messge.
  2. Bounded capacity – finite length of *n* messages
  Sender must wait if link full
  3. Unbounded capacity – infinite length
  Sender never waits

Shared memory vs message passing

- Message passing may be advantageous for exchanging smaller amounts of data since the synchronization overhead is avoided.
- Shared memory can be faster than message passing, particularly for larger amounts of data since no copying is involved. This is true only in cases where the synchronization overhead can be minimized.
- However, in multi-processing (or multicore) systems, research has shown that message passing is more efficient, even for larger blocks of data, due to the cache coherency overhead.

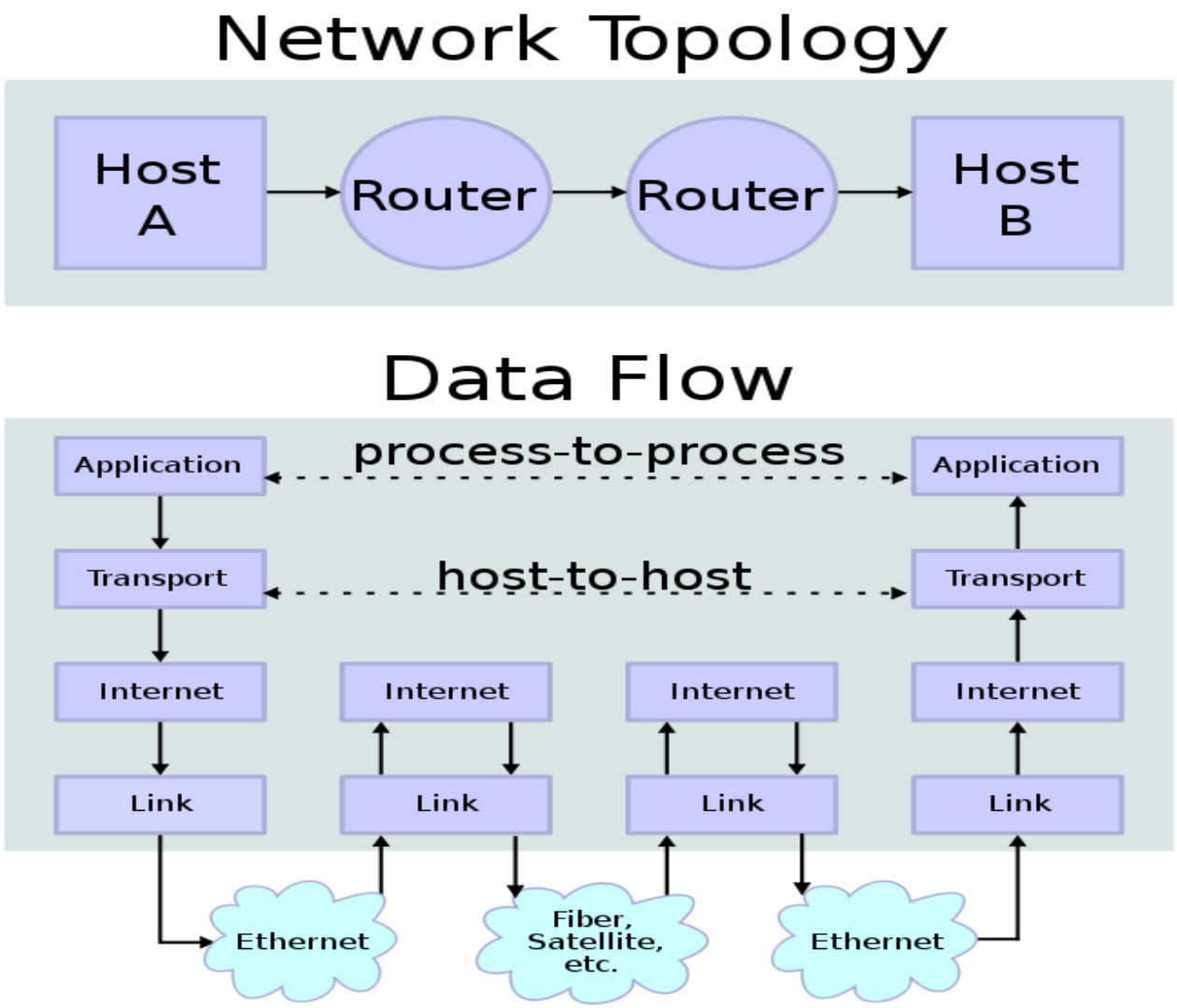
3.6 Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Pipes

The internet layered communication model model

There are two main communication models:

- The internet model
  - 4 layers
  - The most popular
- The Open system interconnect model
  - 7 layers
  - Not as popular today

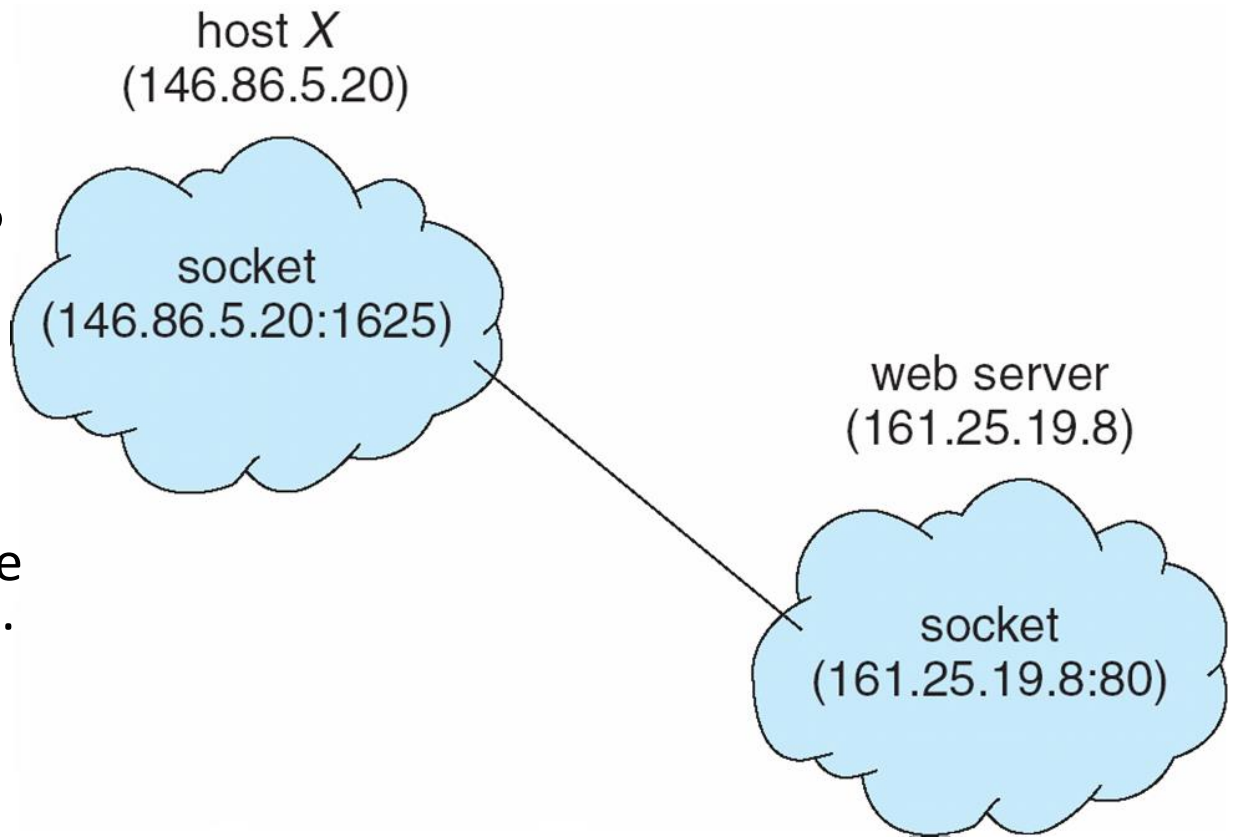


3.6.1 Sockets

- A **socket** is defined as an endpoint for communication
- A socket is the concatenation of an IP address and a **port** number (a number included at start of message packet to differentiate network services on a host).
- Communication consists between a pair of sockets, to form a virtual circuit.
- All ports below 1024 are ***well known***, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running

Socket Communication

- A client process may open a socket by specifying its protocol type. The system assigns the socket the IP address of the system and an arbitrary port number (above 1024, e.g. 1625), and thus the socket or end point is **146.86.5.20:1625**
- The protocol may be:
  - Transmission control protocol **(TCP)**: connection oriented
  - User datagram protocol **(UDP)**: connectionless
- The client process may then connect the socket to a server at **161.25.19.8:80,** i.e. at port 80 of the server, thus forming a virtual circuit or connection.
- e.g. an http/web servers listens to port 80, while an ftp server listens to port 21.



3.6.2 Remote Procedure Calls

- Sockets are a form of low-level communication since they do not specify the data format. Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation (e.g. Oracle (Sun microsystems) RPC uses port 111 (may use TCP or UDP)
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshals** the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language** (**MIDL**)

Remote Procedure Calls (Cont.)

- Data representation handled via **External Data Representation** (**XDL**) format to account for different data representations (i.e. **Big-endian vs little-endian)**
- Remote communication has more **failure scenarios** than local procedure calls due to communication errors (lost or duplicated messages)
  - OS must ensure messages are delivered ***exactly once.***
- There are many different, and **often incompatible, RPC standards** (e.g. Oracle RPC, used for network file systems, Microsoft DCOM remoting, Google Web ToolKit, etc.)
- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server

Execution of RPC