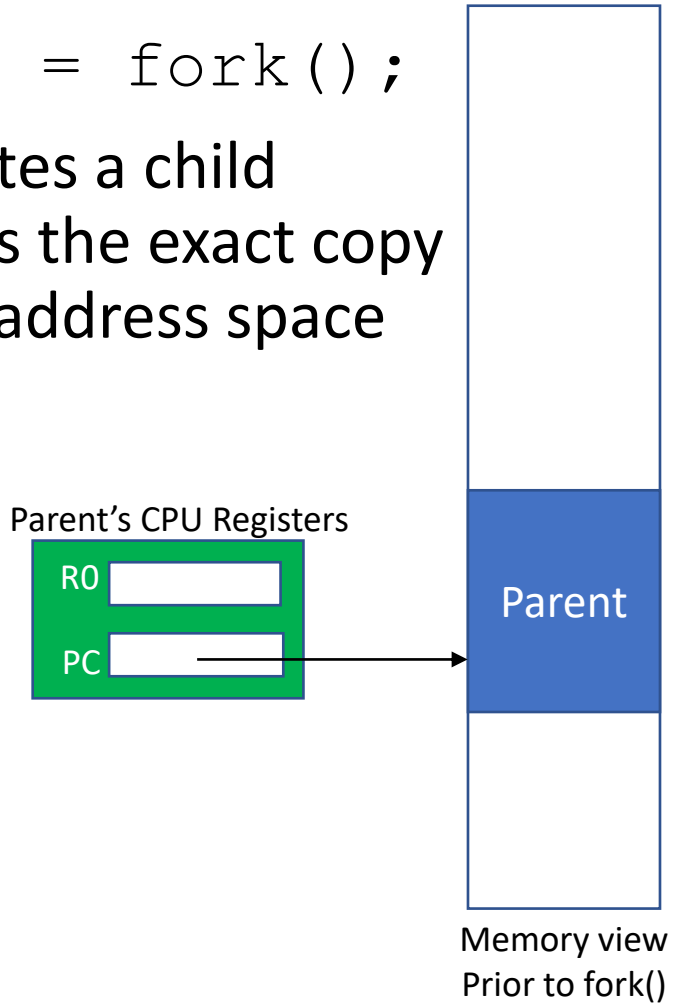
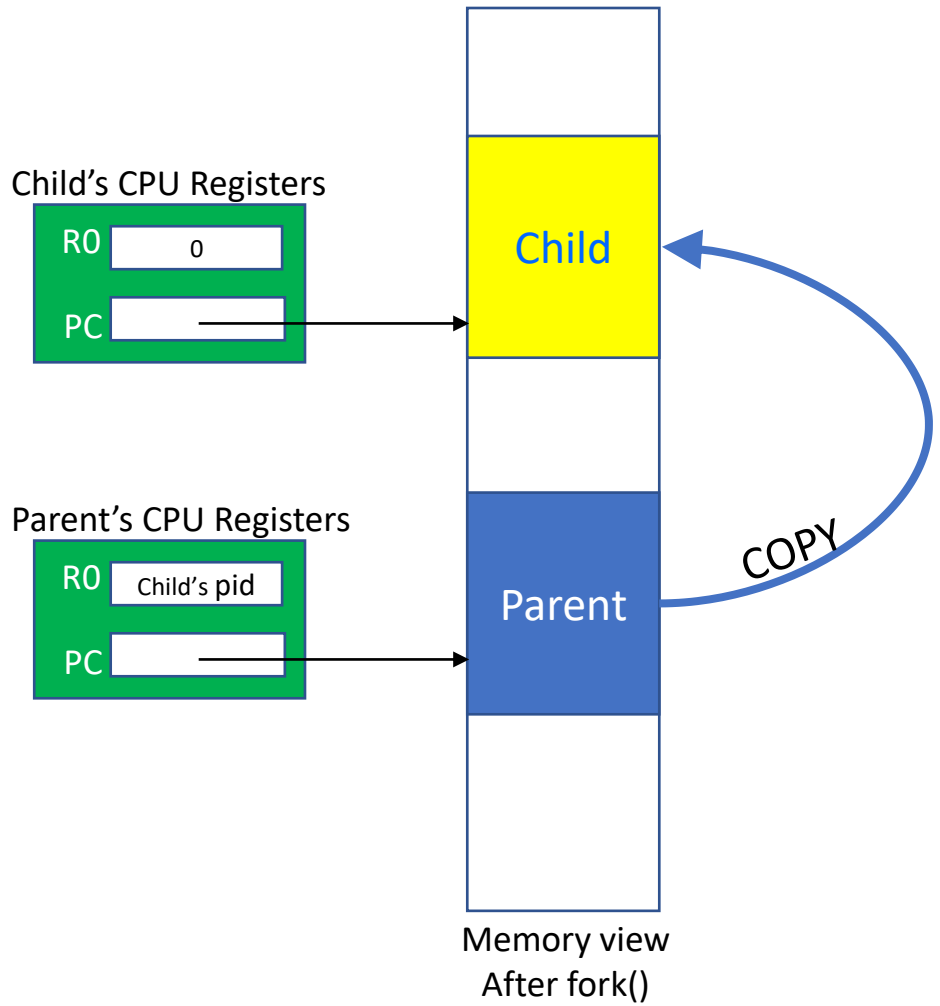
# Creating a separate process in Unix/Linux - cont.

When using fork(), e.g.:

```
int ret = fork();
```

- The kernel creates a child process that has the exact copy of the parent's address space content

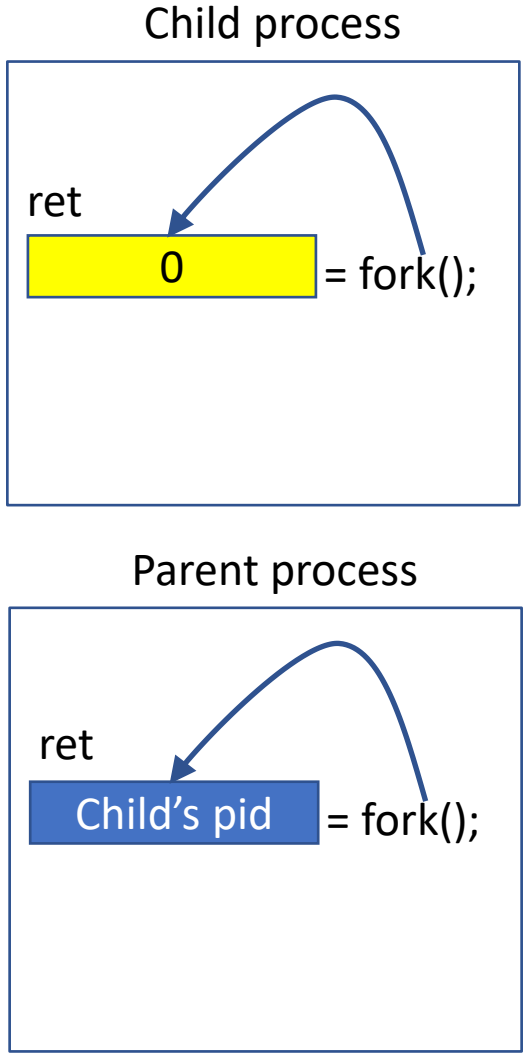


# Creating a separate process in Unix/Linux - cont.

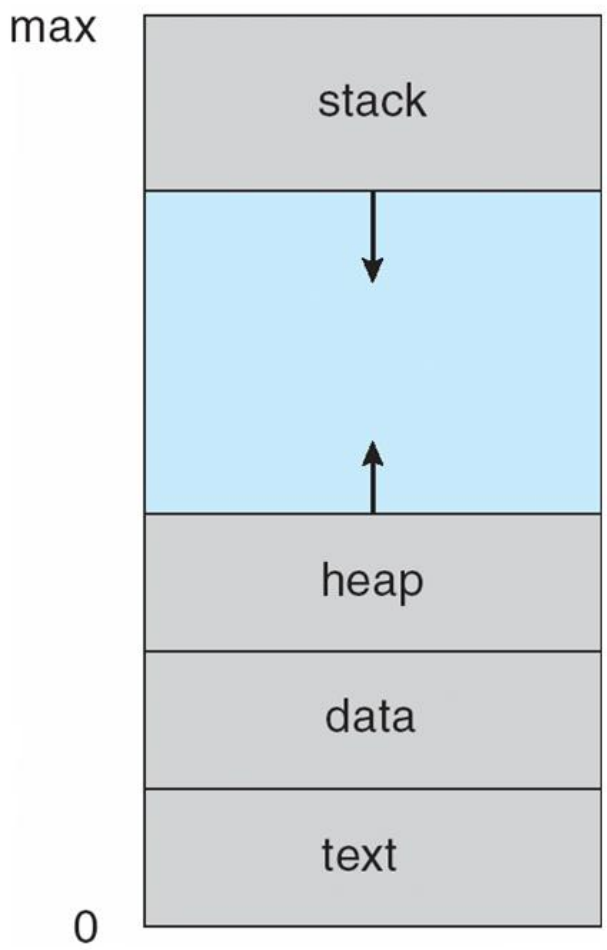
When using "`int ret=fork()`" – cont.

- The parent executes and then invokes the fork() function within the API library
  - which invokes the system call (via a **trap instruction, obviously!**)
- Upon return from the system call, both the parent and the child processes resume at the instruction following the fork() call.
  - That would be the assignment statement!
- Note that generally, each process has its own **address space in memory**. Unless explicitly requested, the kernel ensures that no process infringes on the address space of another process



# Creating a separate process in Unix/Linux

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```



# Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call. This causes:
  - Returns status data from child to parent (via the parent calling **`wait()`**)
  - Process' resources are deallocated by operating system
  - In Linux, `exit` usually takes one parameter indicating an error if non-zero.
  - `exit` is called implicitly upon return from the main routine of a program.
- Parent may terminate the execution of children processes using the **`abort()`** system call (`TerminateProcess()` in windows). Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

# Process Termination

- Some operating systems do not allow a child to exist if its parent has terminated. Hence, in such systems, if a process terminates, then the OS terminates all its children.
  - **cascading termination** - All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.

  (this is not the case in Linux)
- The parent process may wait for termination of a child process by using the **`wait()`** system call. The call returns status information and the pid of the terminated process

```
int status;
pid = wait(&status);
```

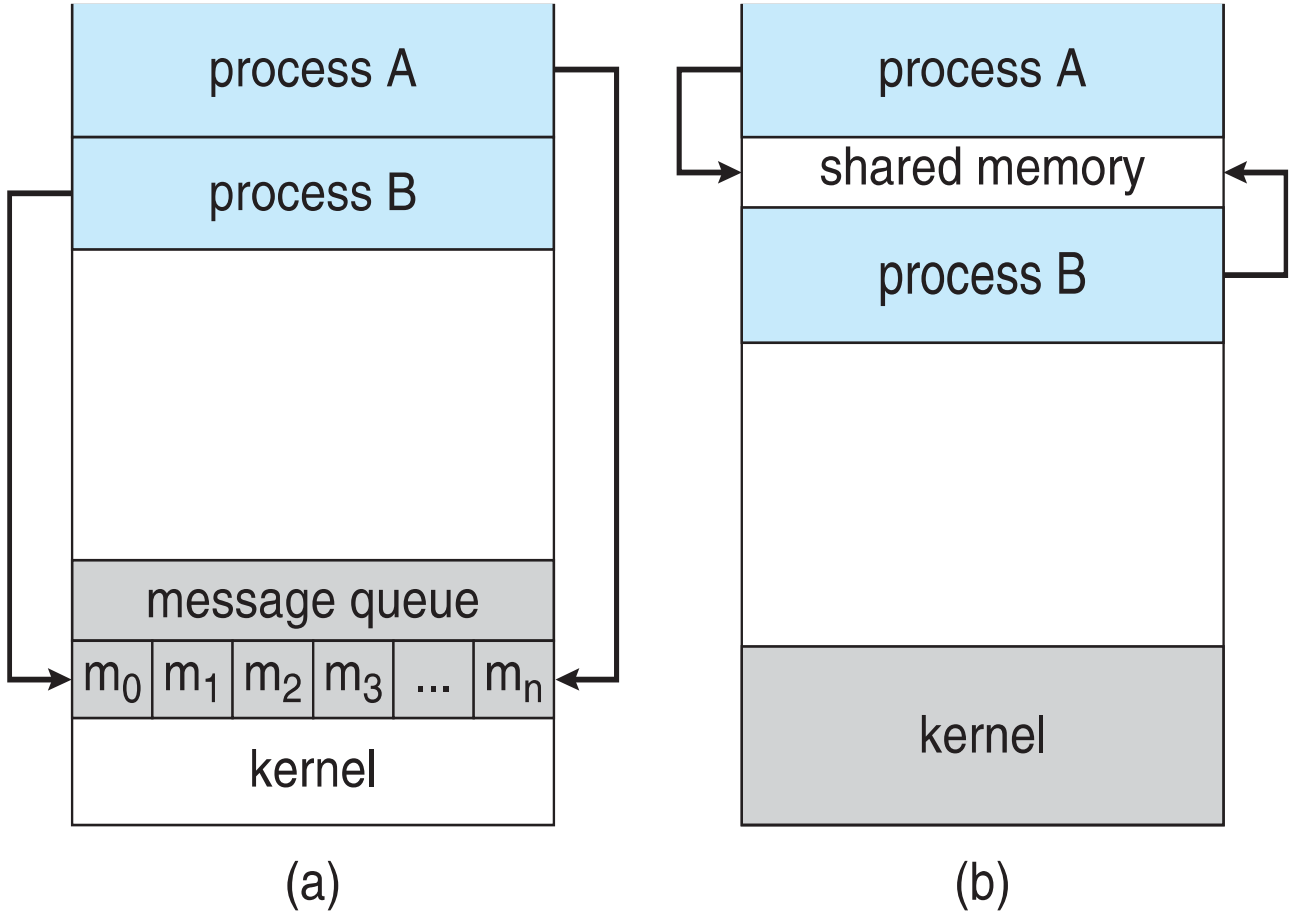
- When a process exits, its resources are deallocated
  - except its entry in the process table (containing the exit status).
  - Only after the parent invokes the wait() function which reads that status that its entry in the process table is released.
  - Till then, the terminated child process is a **zombie.**
- If a parent terminated before the child (i.e. without invoking **`wait)`**, the running child process is an **orphan** (**if allowed by OS,** e.g. Linux) and its new parent becomes the init process (whose PID is 1).
  - The init process periodically invokes **`wait`** in order to release orphan zombie processes.

# 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 communicate using message passing (via Unix pipes or TCP/IP sockets).

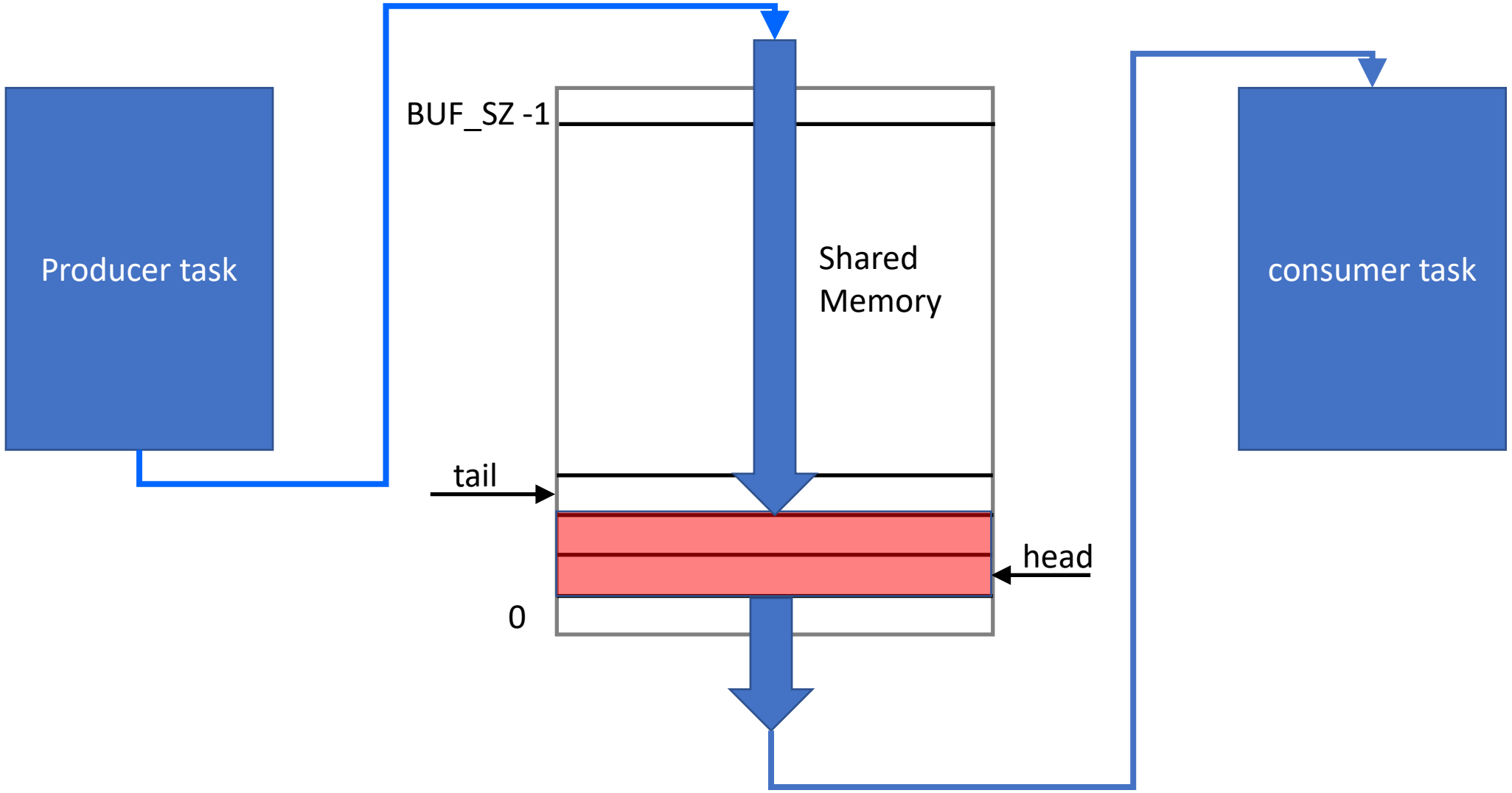
# 3.4.1 Shared memory systems

- 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 be solved using shared memory.
  - It may also be solved using message passing communication as we shall see later.
- 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

# The producer-consumer problem with shared bounded buffer

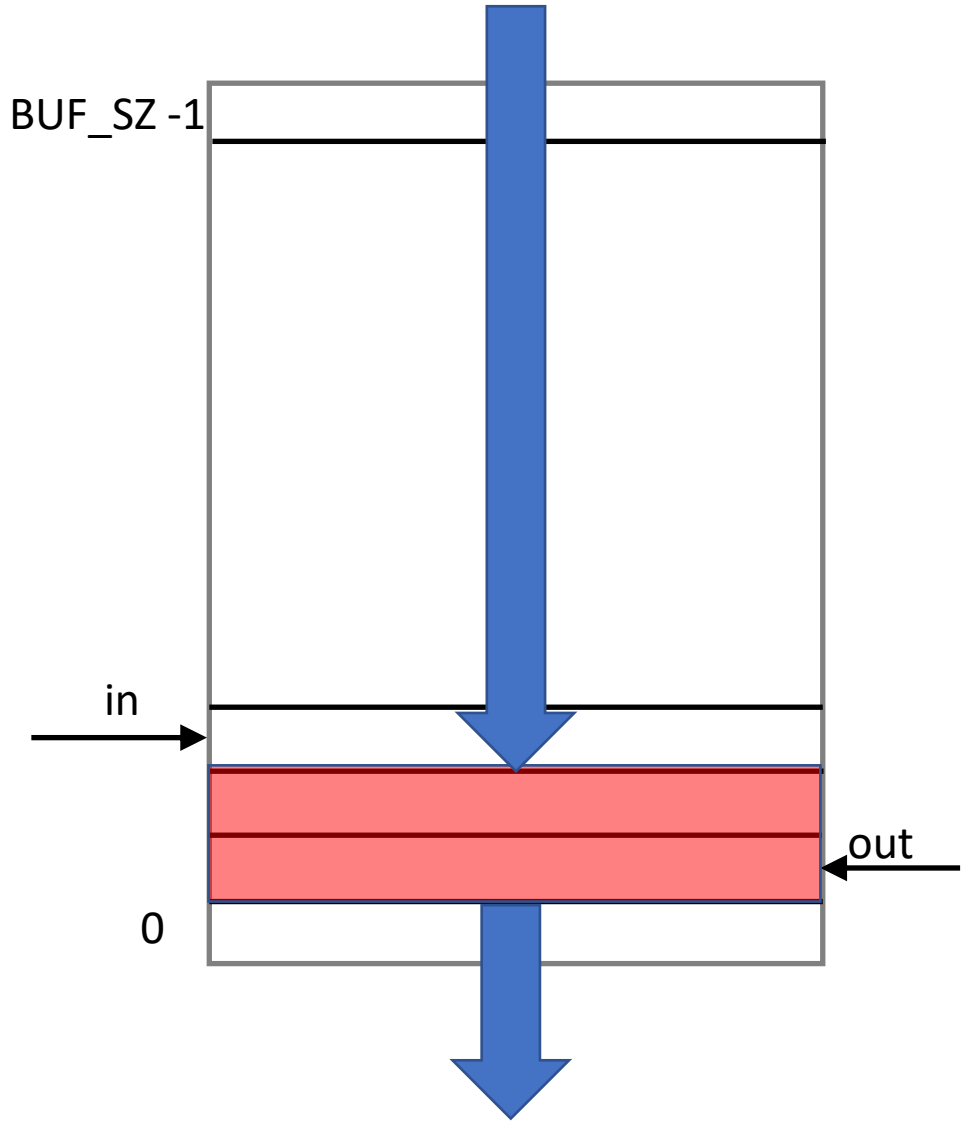


# Bounded-Buffer – Shared-Memory Solution

- Shared data:

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

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

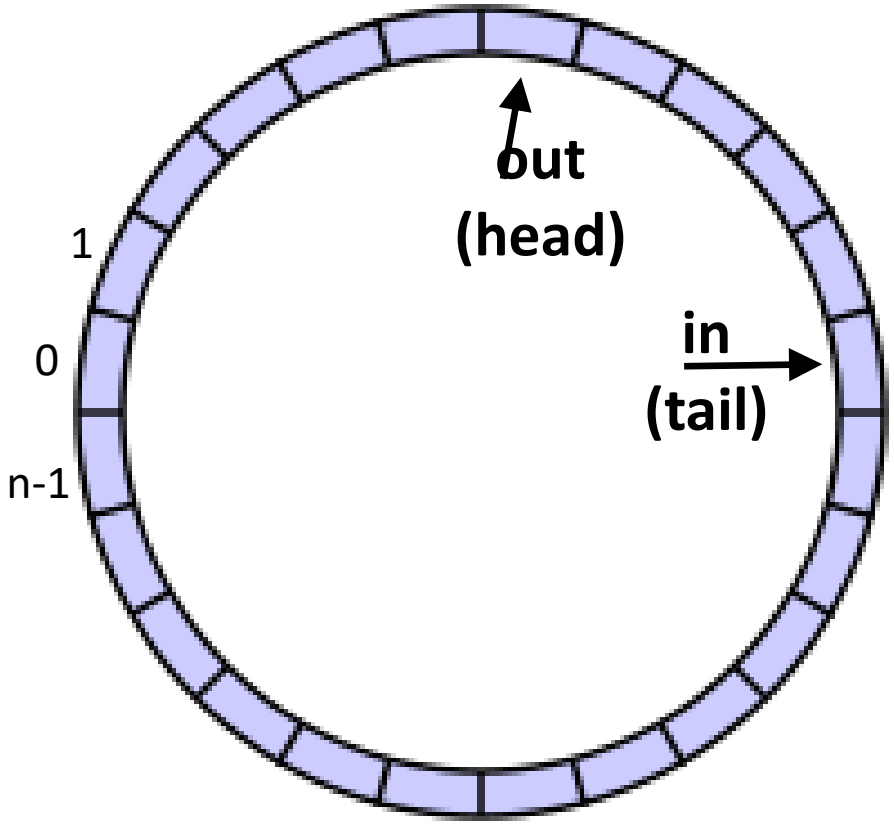
- Buffer needs to be administered and used as a FIFO or Queue



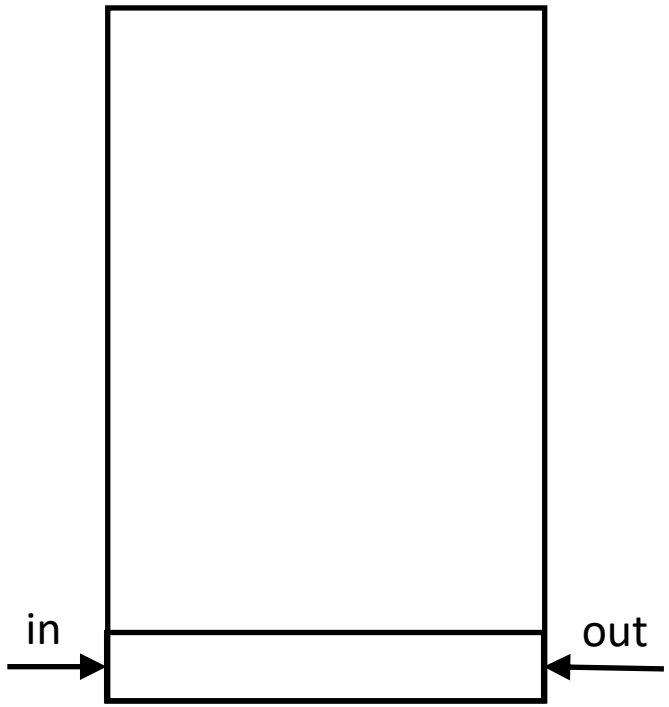
# Bounded-Buffer – Shared-Memory Solution

- Shared data:

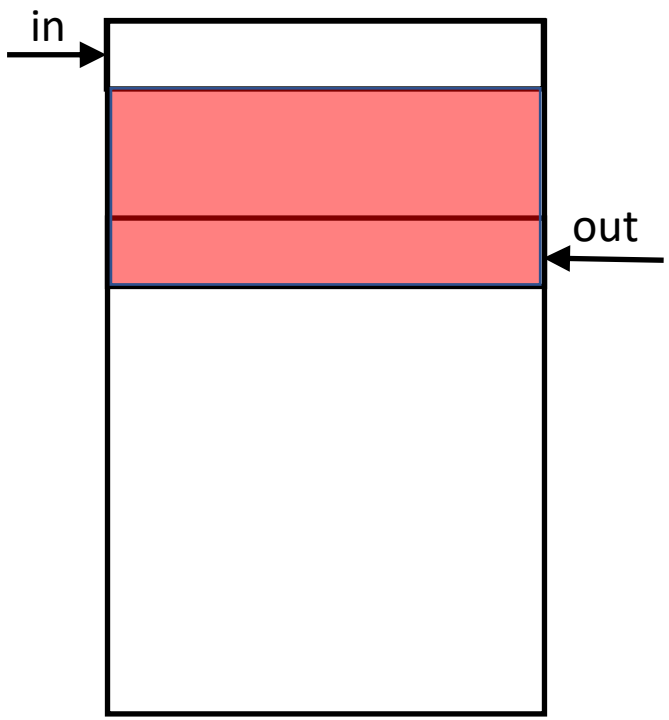
```
#define BUF_SZ 5
typedef struct {
  . . .
} item;

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



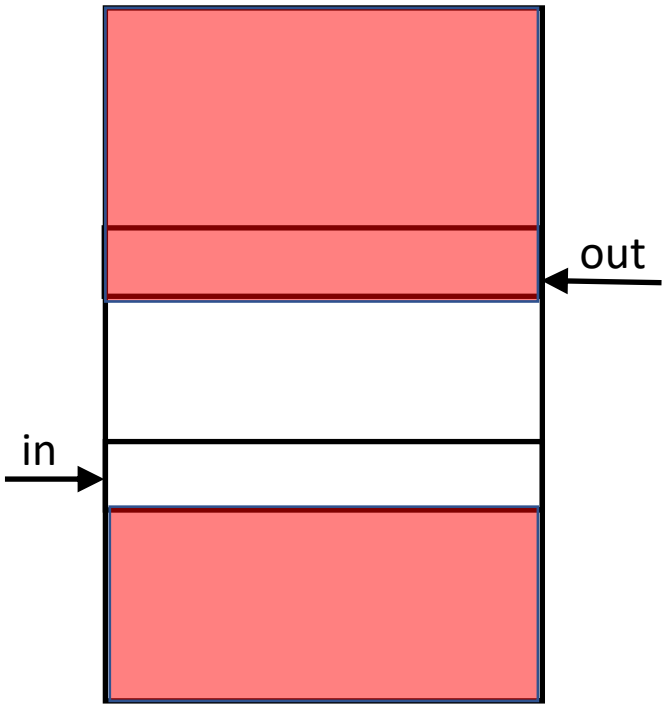

- Buffer needs to be administered and used as a FIFO or Queue



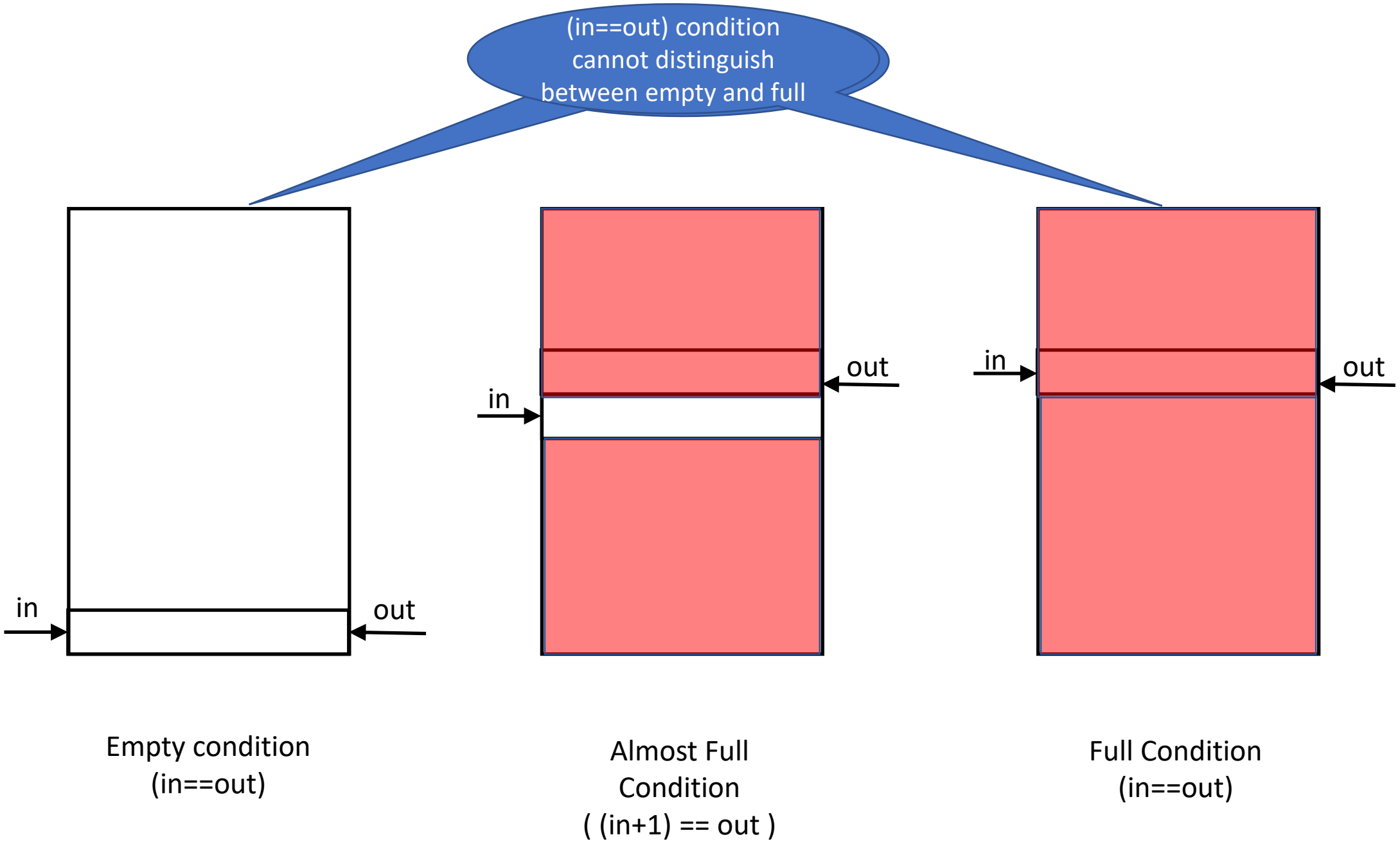

Initial state




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


(in==out) condition
cannot distinguish
between empty and full
in
out
in
out
in
out
Empty condition
(in==out)
Almost Full
Condition
( (in+1) == out )
Full Condition
(in==out)

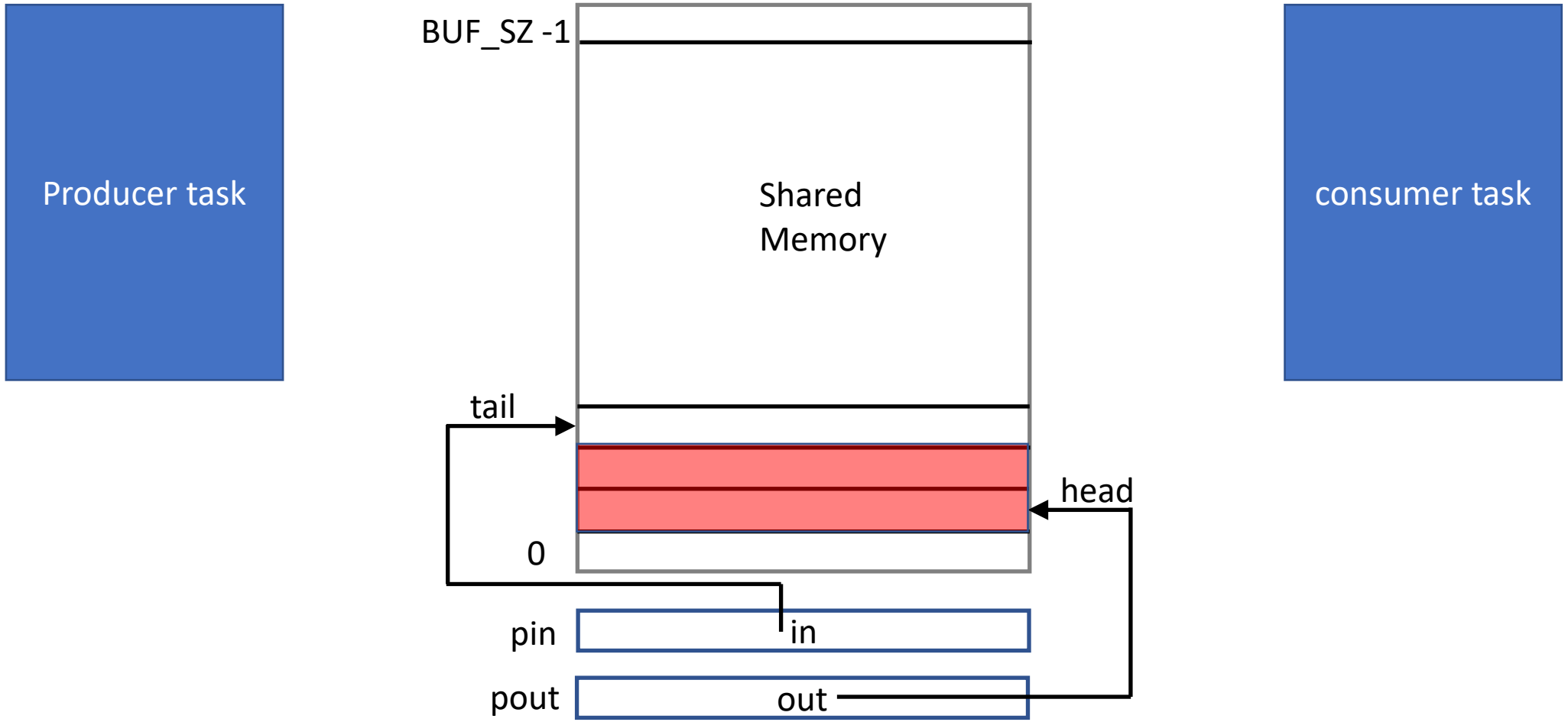
# Producer-consumer **example**

- Assume a producer task wishes to send characters, one at a time to a consumer task, e.g. the message below

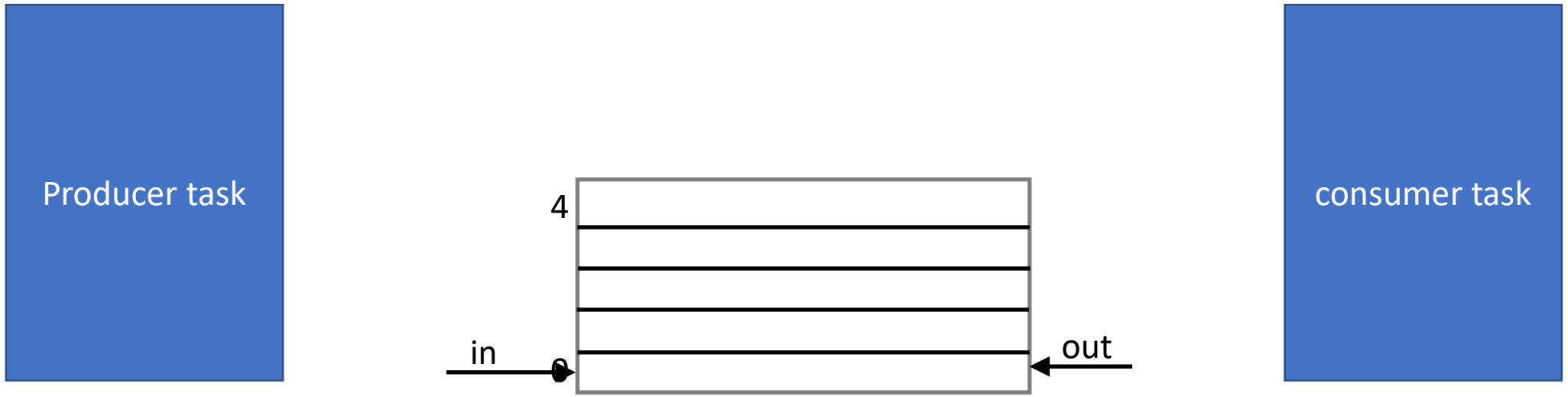
  `"Hello world!"`

- Then a few steps need to be followed:
  - Create shared memory for the ring buffer, and get its address `buf` (whose type is `char*`),
  - Create shared memory for the head index, and get its address, `pout` (`int*`)
  - Create shared memory for the tail index, and get its address, `pin` (`int*`).
  - The producer uses an enqueue function and enqueues each of the characters into the shared buffer.
  - The consumer uses a dequeue function and dequeues each of the characters and prints it to the screen.

# Producer-consumer example – shared memory view

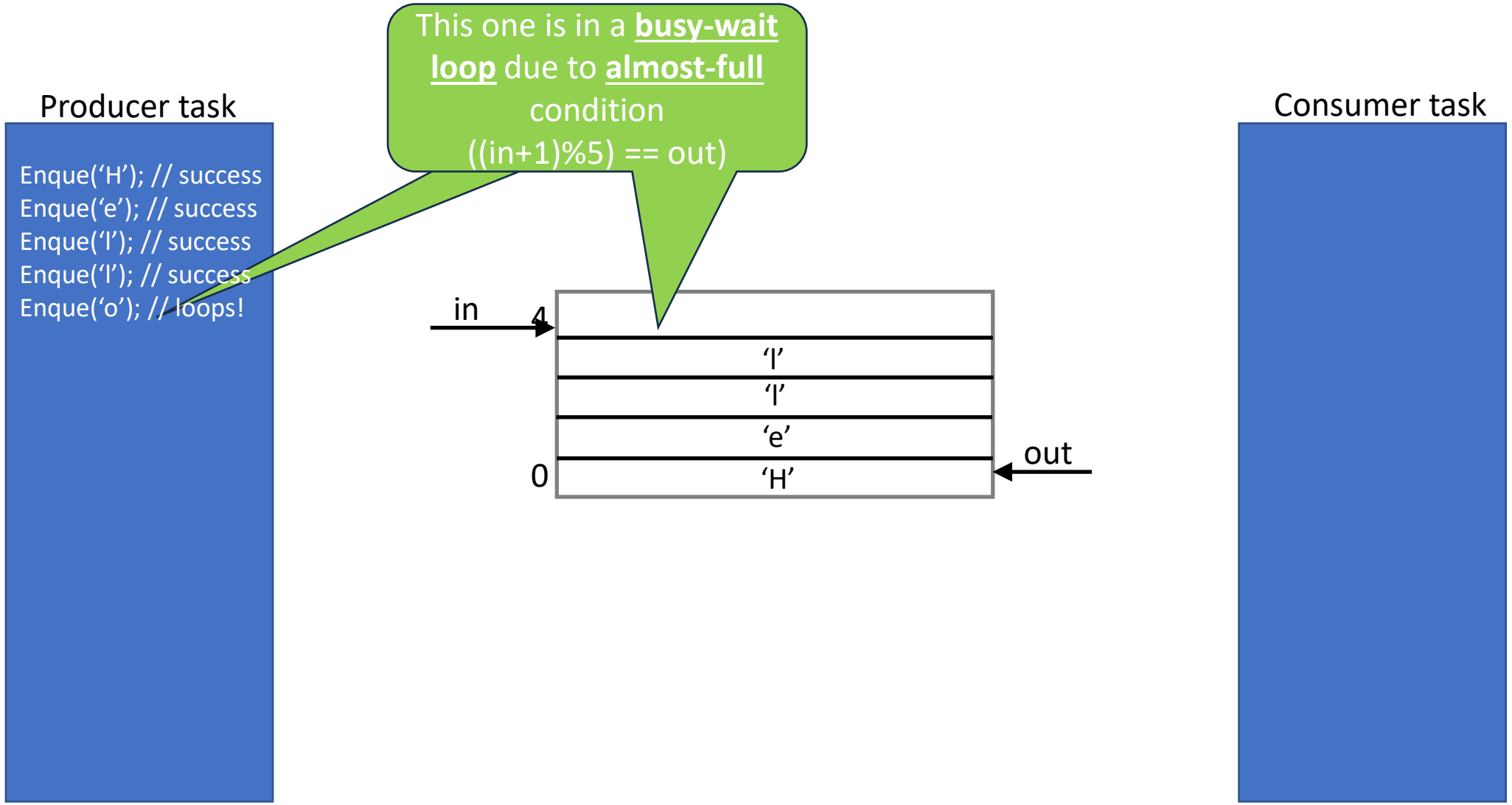


# Producer-consumer **example** – buffer size=5




```
void Enqueue(char c){
   // wait till "almost-full" is false
   while (((*pin + 1) % BUF_SZ) == *pout);
   buf[(*pin)] = c;
   *pin = (*pin + 1) % BUF_SZ;
}
```

# Producer-consumer **example** – buffer size=5



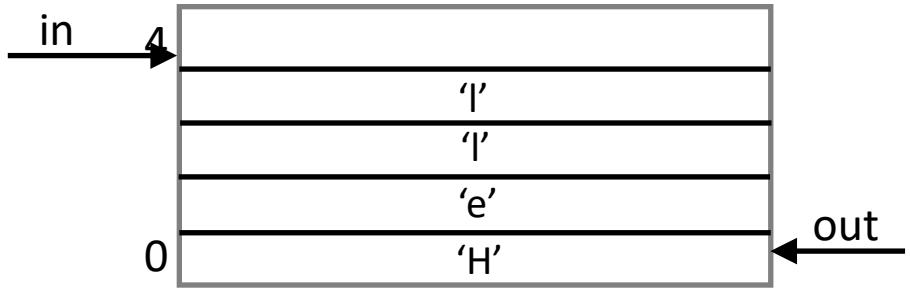
# Producer-consumer **example** – buffer size=5

At some point the producer task will be preempted

Producer task

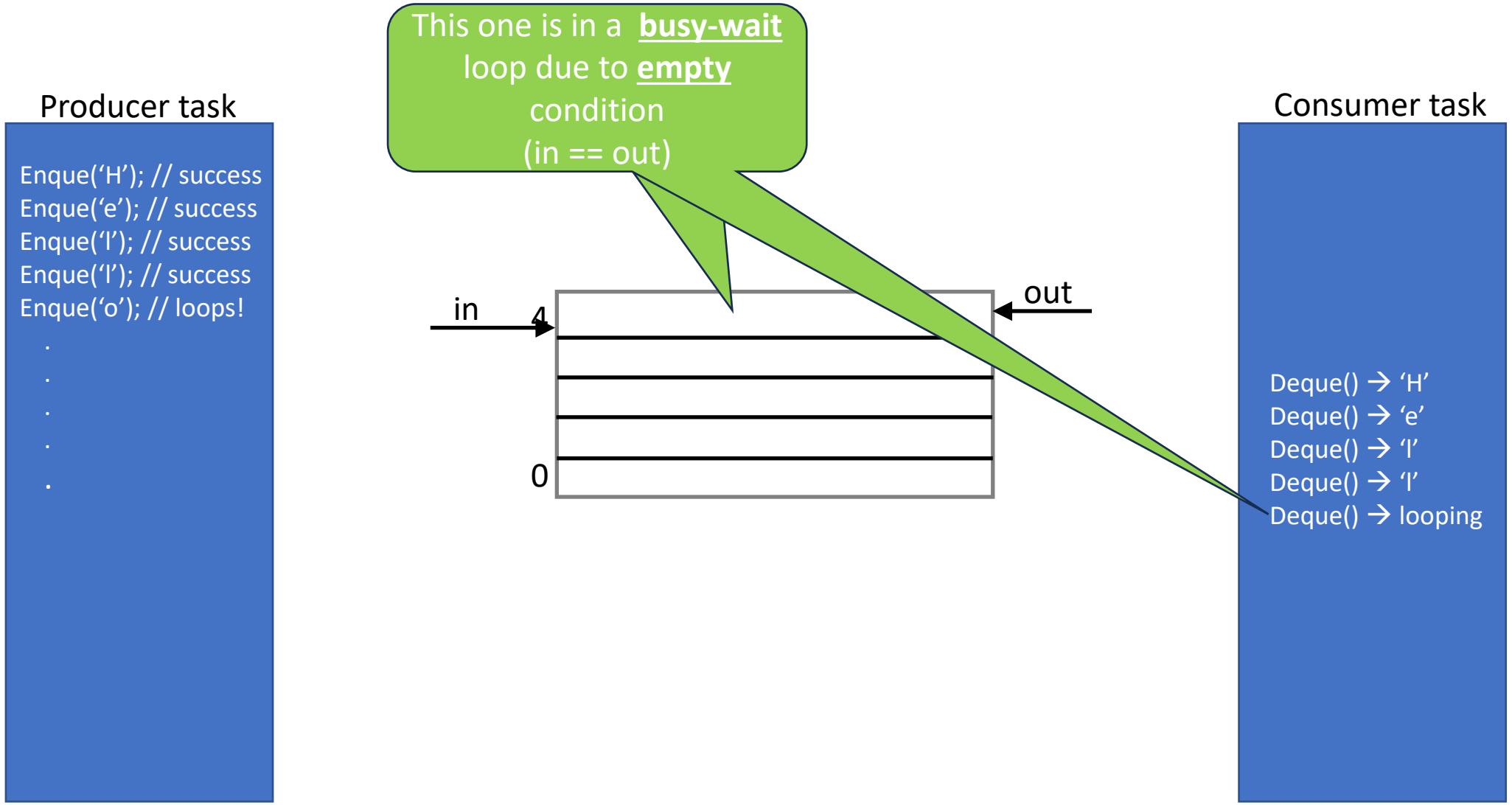
```
Enque('H'); // success
Enque('e'); // success
Enque('l'); // success
Enque('l'); // success
Enque('o'); // loops!
.
.
.
.
.
```

Consumer task

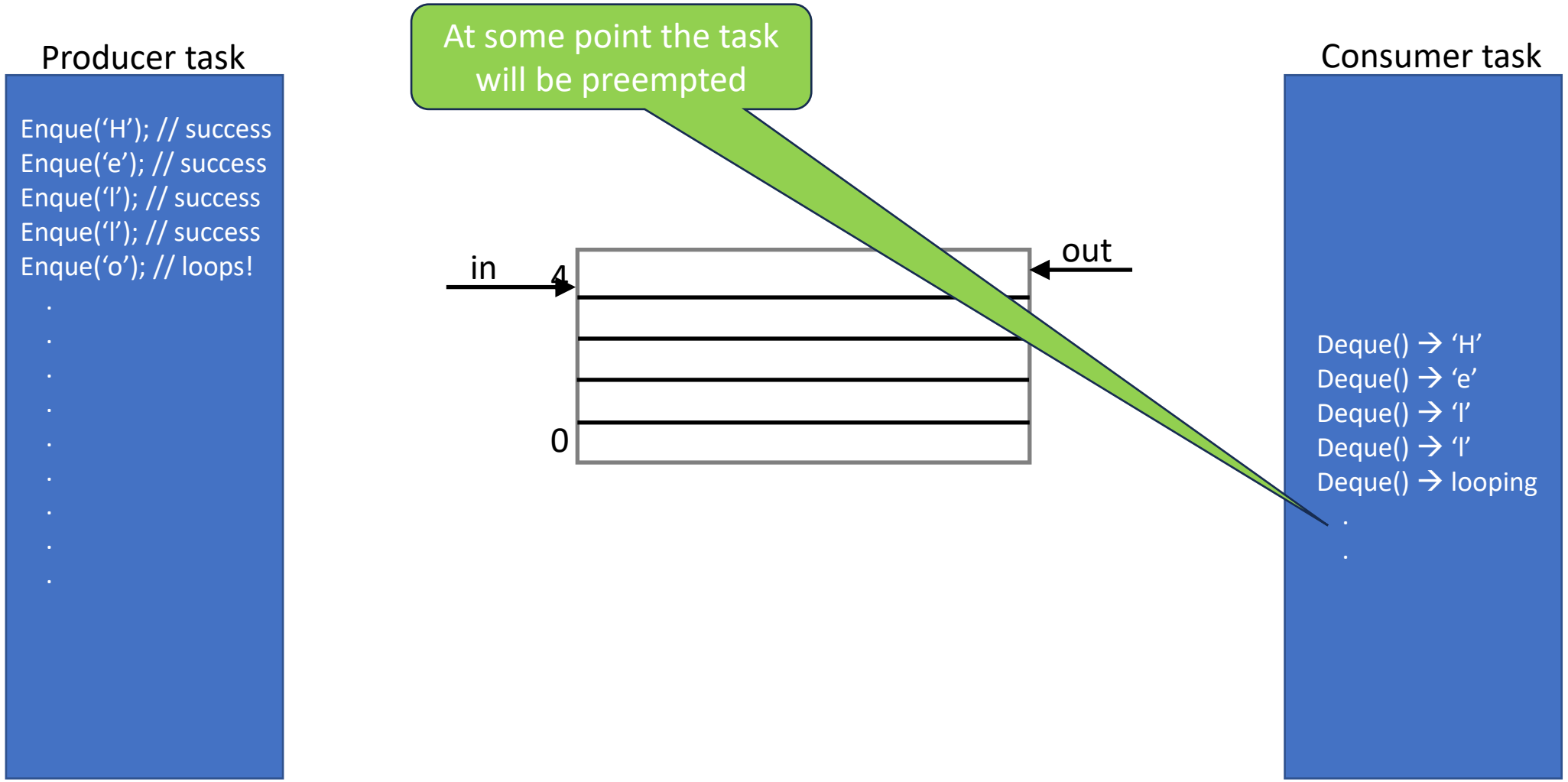



```
char Deque()[
   // wait for "empty" to be false
   while (*pin == *pout);
   char temp = buf[(*pout)];
   *pout = (*pout + 1) % 5;
   return temp;
}
```

# Producer-consumer **example** – buffer size=5



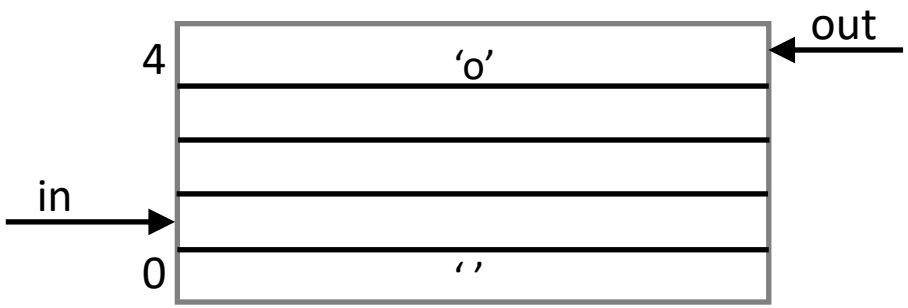
# Producer-consumer **example** – buffer size=5



# Producer-consumer **example** – buffer size=5

Producer task

```
Enque('H'); // success
Enque('e'); // success
Enque('l'); // success
Enque('l'); // success
Enque('o'); // loops!
   .
   .
   .
   .
   .
   .
   .
   .
   .
// Enque('o') now
// succeds
Enque(' '); // success
   .
   .
```



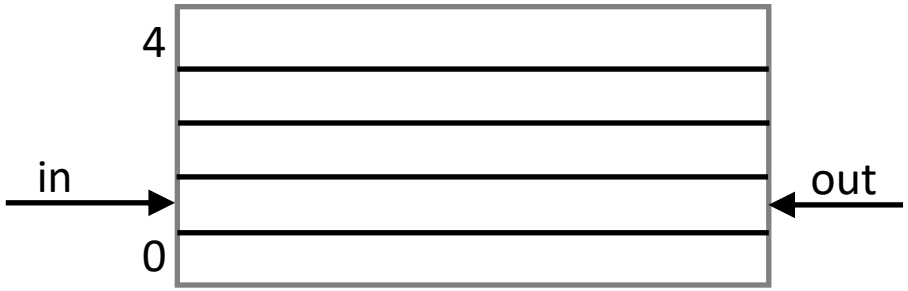

Consumer task

```
Deque() → 'H'
Deque() → 'e'
Deque() → 'l'
Deque() → 'l'
Deque() → looping
   .
   .
   .
```

# Producer-consumer **example** – buffer size=5

Producer task

```
Enque('H'); // success
Enque('e'); // success
Enque('l'); // success
Enque('l'); // success
Enque('o'); // loops!
   .
   .
   .
   .
   .
   .
   .
   .
   .
// Enque('o') now
// succeds
Enque(' '); // success
   .
   .
```



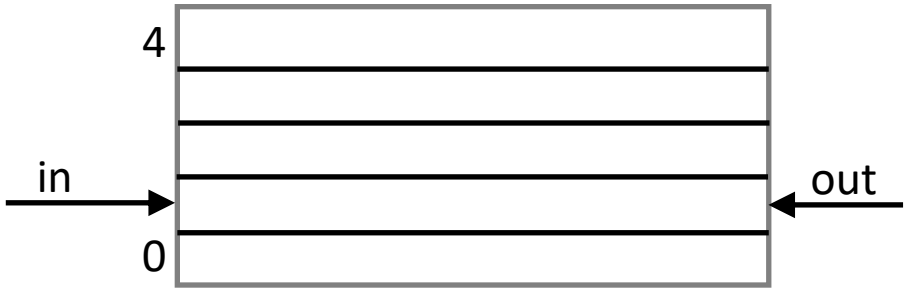

Consumer task

```
Deque() → 'H'
Deque() → 'e'
Deque() → 'l'
Deque() → 'l'
Deque() → looping
   .
   .
   .
   .
   .

→ 'o'
Deque() → ' '
```

# Producer-consumer **example** – buffer size=5

Producer task

```
Enque('H'); // success
Enque('e'); // success
Enque('l'); // success
Enque('l'); // success
Enque('o'); // loops!
   .
   .
   .
   .
   .
   .
   .
   .
   .
// Enque('o') now
// succeds
Enque(' '); // success
   .
   .
```




Consumer task

```
Deque() → 'H'
Deque() → 'e'
Deque() → 'l'
Deque() → 'l'
Deque() → looping
   .
   .
   .
   .
   .
   .
→ 'o'
Deque() → ' '
Deque() → loops
```

# 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);
```

  - Unrelated processes (i.e. ones without a parent-child relationship) 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 memory 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.

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t
  offset);
```

- Now the process could write to the shared memory, e.g.

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

# 3.5 Examples of IPC Systems – POSIX shared memory – cont.

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

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

- There are 3 permissions per owner (of a node on the filesystem):
  - **Read permission**
  - **Write permission**
  - **Execute permission**
  - Specified using an octal numbering system
    - 06 → an octal 6 → which is 110 in binary → read, write, but NOT execute permission
    - 0x61 → hexadecimal 61
    - 6 → decimal 6

# POSIX – cont.

- In Unix/Linux, the shared memory is abstracted as a virtual file 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 example that sends 2 c-strings

```
#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 Producer example that sends 2 c-strings

```
#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
  - Can also solve the producer-consumer problem (in a much easier way!)
- 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 a link unidirectional or bi-directional?
    - Is the size of a message that the link can accommodate fixed or variable?
    - What is the capacity (i.e. total storage size) of a link?

# Message Passing (Cont.)

- Implementation of communication link
  - On the physical level:
    - **Via shared memory, administered by the kernel**
    - 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 e.g ordinary pipes or bidirectional e.g. named pipes) 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 addressed 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

**producer**

```
void Enqueue(item x){
    send(x);
}
```

**consumer**

```
void Dequeue(item& y)
    receive(y);
}
```

# 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 message.
  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 Message passing Communications

- Pipes
- Sockets
- Remote Procedure Calls

# 3.6.3 Pipes

- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
    - In the case of bidirectional 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
- 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
- They are referred to as ordinary pipes in Unix/Linux, but Windows calls them **anonymous pipes.**

# Ordinary Pipes – example

```c
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
   char write_msg[BUFFER_SIZE] = "Hello, world!";
   char read_msg[BUFFER_SIZE];
   int fd[2];
   pid_t pid;

   /* create the pipe */
   if (pipe(fd) == -1) {
      fprintf(stderr, "Pipe failed");
      return -1;
   }
```

```c
/* fork a child process */
pid = fork();

if(pid<0){ /* error occured */
   fprintf(stderr, "Fork Failed");
   return 1;
}

else if (pid > 0) {/* parent process* /
   /* close the read end since we are the producer */
   close(fd[READ_END]);

   /* Write to the pipe */
   write(fd[WRITE_END], write_msg, strlen(write_msg) + 1);

   /* close the write end, now that we are done */
   close(fd[WRITE_END]);
}
```

```
    else { /* child process*/
        /* close the write end since we are the consumer */
        close(fd[WRITE_END]);

        /* read from the pipe and print the message to screen */
        read(fd[READ_END], read_msg, BUFFER_SIZE);
        printf("read %s\n", read_msg);

        /* close the read end now that we are done */
        close(fd[READ_END]);
    }
    return 0;
}
```

# 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

# Named Pipes in Unix-like systems

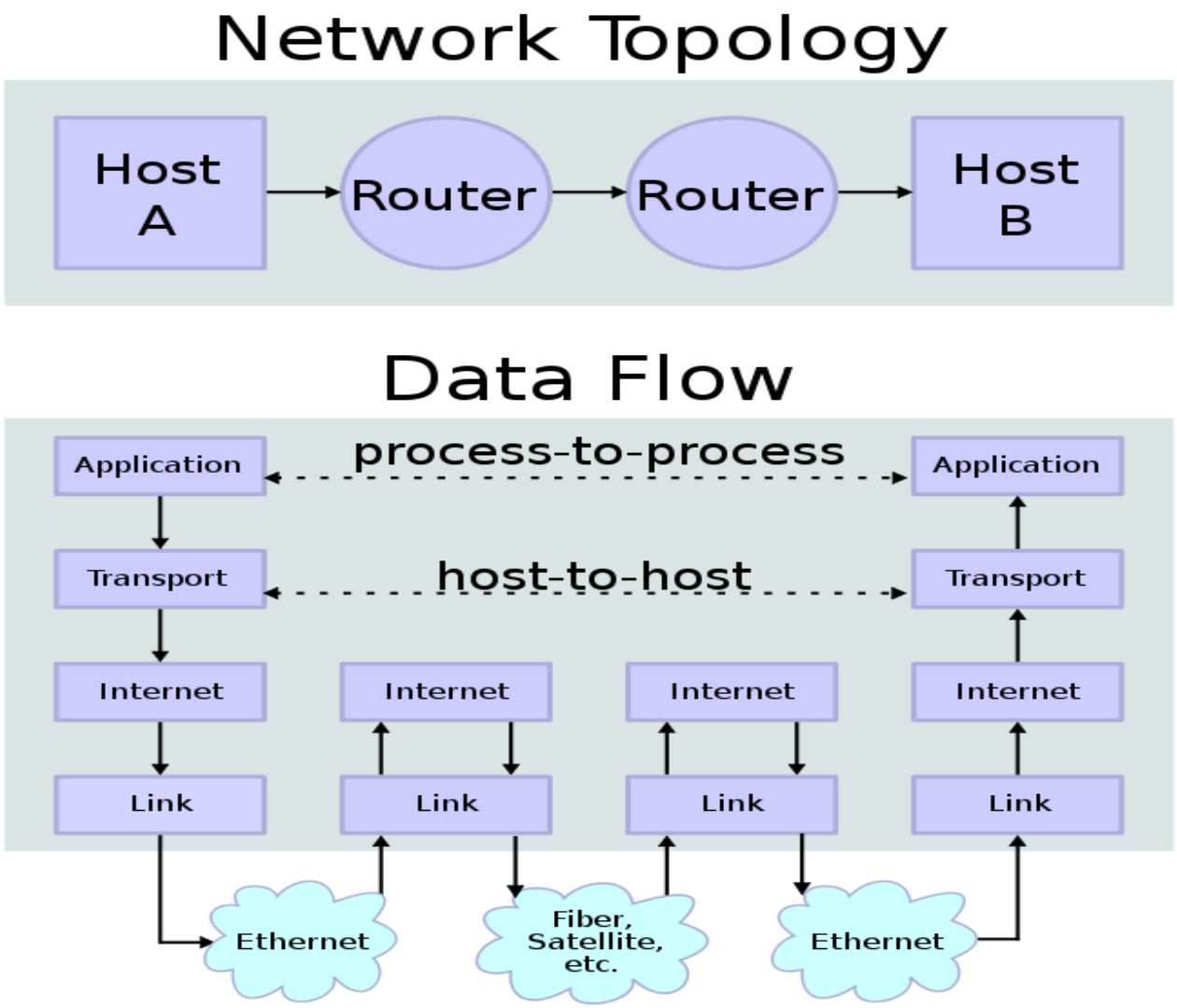
- Unix supports named pipes via the **`mkfifo()`** followed by **`open()`** library calls:
  - Note that the two (or more) communicating processes need to use the same pipe name.
  - The default fifo behavior is blocking, however the open flag **O_NONBLOCK** controls whether blocking or non-blocking.
  - In **blocking calls**, both (all) processes are blocked till at least one open for read and one open for write have occurred.
  - **In non-blocking calls**, a process opening the FIFO in read-only always succeeds, whereas a process opening the FIFO in write-only fails if no other process already opened the FIFO for reading.
  - **Whether blocking or non-blocking**, If one process opens the FIFO with read-write mode, then the process will not block or fail.
  - After the FIFO is opened, normal file operations can be used to read and write into the FIFO.

```
int mkfifo(const char *pathname, mode_t mode);
```

# 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

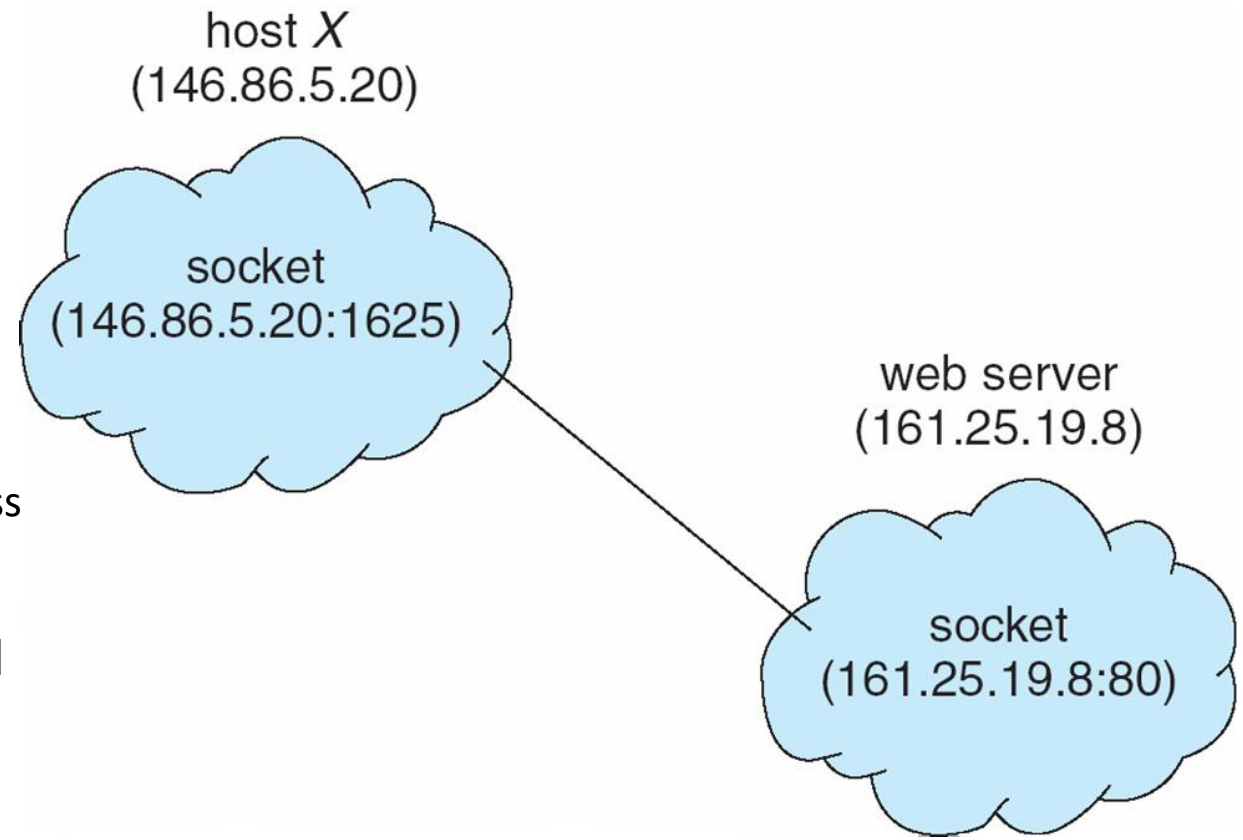
- A **socket** is defined as an endpoint for communication
- A socket is the concatenation of an **IP address + a port number** (a number included at start of message packet to differentiate network services on a host).
- Communication consists between a pair of sockets, to form a virtual circuit.

You need to memorize this

- Port numbers are a **16-bit values**
- 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:
  - An IP address (equals IP of the client or host)
  - An arbitrary port number (above 1024, e.g. 1625)
  - e.g. 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.
- Port number examples:
  - http/web servers listens on port 80,
  - Secure web server (SSL) listens on 443
  - ftp server listens on port 21.
  - Un-encrypted smtp mail server: 25
  - Encrypted smtp mail server: 465



# 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

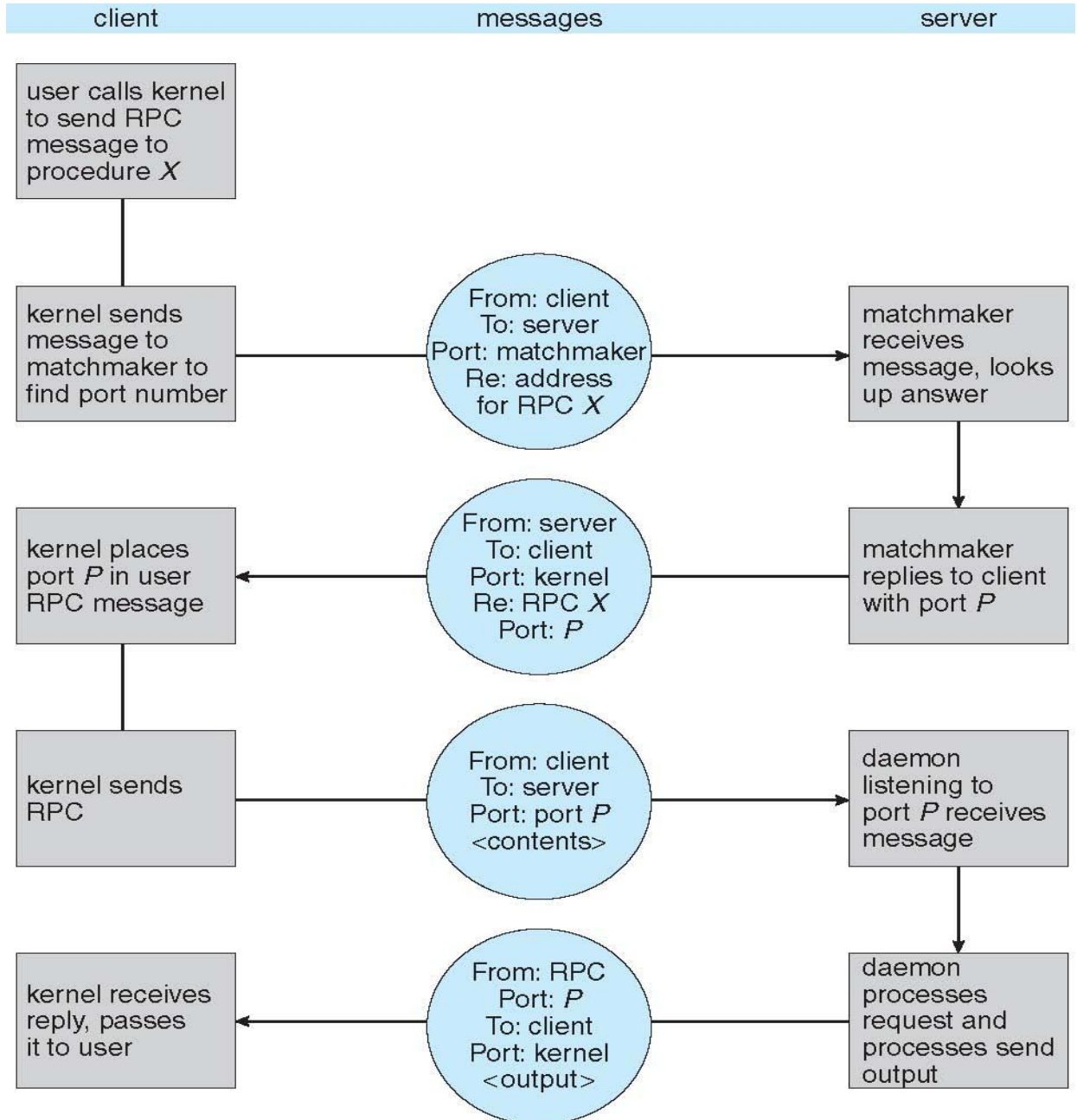
# Remote Procedure Calls (Cont.)

- Stubs need to address:
  - 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



# Local Procedure calls in Windows

- Message-passing centric via **advanced local procedure call (LPC)** facility
  - Exclusively used by windows OS components. Not accessible to user-mode applications (i.e. not available on public windows API).
  - Only works between kernel processes/threads on the same system
  - Uses ports (= mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - The client opens a handle to the subsystem's (server) **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 Calls in Windows