

Inter Process Communication

Girish S Kumar

Operating System

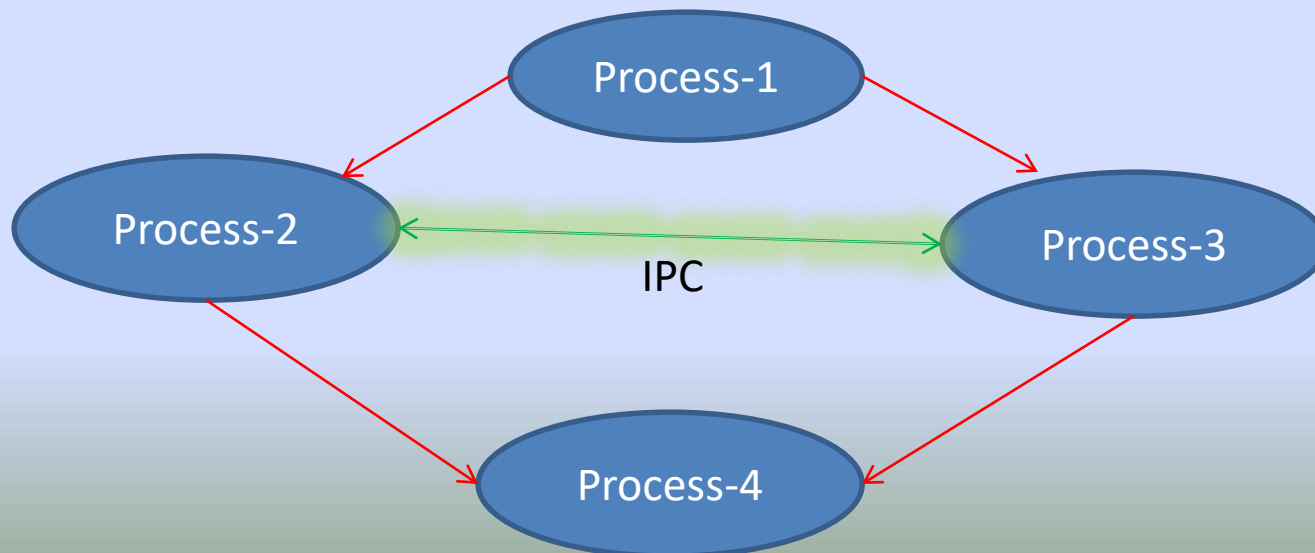
Services to applications

- ✓ Program Execution
- Inter-process Communication
- I/O operations

What is IPC

In the previous section we saw how a program execution takes places discussed the concepts of Process and threads

For performing a task through a computer, we need the task to be broken down into small steps that can be executed either in series or parallel. When task is broken down into small tasks that can be executed in parallel, there is a need to exchange information between the parallel task or synchronize between them.



Methods for IPC

Most of the Operating systems has implemented the following methods

- Signals
- Pipe
- Sockets
- Message Queues
- Semaphore
- Shared Memory

Signals

Signals are asynchronous events (unexpected events) that are sent to an application/process.


Example of unexpected events are

1. When you press CTRL-C the program aborts this because the app get a Signal to exit and OS Kernal stops the execution (SIGINT)
2. Core Dump and program aborts
3. A timer set by app expires


How do we handle Signals

When the signal occurs or an unexpected event arrives , the process has to tell the kernel what to do with the event. A process can three options to handle a signal

The signal can be ignored, like division by zero error etc, without causing an adverse impact to results. Provided the Signal is not fatal



The signal can be “caught” so that user defined function is executed, provided the signal is non fatal. The Kernel will execute the function/routine user has provided



Default Action: Every Signal has a default action , this can process it, terminate the program or ignore

Signal That cannot be Caught, Ignored or take the default action

There are two signals SIGKILL and SIGSTOP which can never be ignored because These are signal generated by OS when root user or the kernel wants to kill or stop any process in any situation .

The default action of these signals is to terminate the process.

How write signal handlers

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>

void my_handler_for_sigint(int signumber)
{
    char ans[2];
    if (signumber == SIGINT)
    {
        printf("received SIGINT\n");
        printf("Program received a CTRL-C\n");
        printf("Terminate Y/N : ");
        scanf("%s", ans);
        if (strcmp(ans,"Y") == 0)
        {
            printf("Existing ....\n");
            exit(0);
        }
        else
        {
            printf("Continung ..\n");
        }
    }
}
```

```
int main(void)
{
    /* Registering the Signal handler */
    if (signal(SIGINT, my_handler_for_sigint) ==
        SIG_ERR)
        printf("\ncan't catch SIGINT\n");

    // Process Loops
    while(1)
        sleep(1);
    return 0;
}
```


User Defined Signals

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>

void sig_handler(int signo)
{
    if (signo == SIGUSR1)
        printf("received SIGUSR1\n");

    else if (signo == SIGKILL)
        printf("received SIGKILL\n");

    else if (signo == SIGSTOP)
        printf("received SIGSTOP\n");

    else if (signo == SIGQUIT)
        printf("received SIGQUIT\n");
}
```

```
int main(void)
{
    if (signal(SIGUSR1, sig_handler) == SIG_ERR)
        printf("\ncan't catch SIGUSR1\n");
    if (signal(SIGKILL, sig_handler) == SIG_ERR)
        printf("\ncan't catch SIGKILL\n");
    if (signal(SIGSTOP, sig_handler) == SIG_ERR)
        printf("\ncan't catch SIGSTOP\n");

    /* PRESS Ctrl + \ */
    if (signal(SIGQUIT, sig_handler) == SIG_ERR)
        printf("\ncan't catch SIGQUIT\n");

    // A long long wait so that we can easily issue
    a signal to this process
    while(1)
        sleep(1);
    return 0;
} /* kill -USR1 PID */
```

How do Threads handle Signals

Every thread has its own private signal mask(a mask that defines which signals are deliverable) but the way signal disposition is done is shared by all the threads in the application. This means that a disposition for a particular signal set by a thread can easily be overruled by some other thread.

For example, a thread A can choose to ignore a particular signal but a thread B in the same process can choose to catch the same signal by registering a callback function to the kernel. In this case the request made by thread A gets overruled by thread B's request.

Threads have their separate signal mask which can be manipulated using [pthread_sigmask\(2\)](#) similar to [sigprocmask\(2\)](#), so such signal is not delivered to a thread that has this signal blocked. It's delivered to one of threads in the process with this signal unblocked. It's unspecified which thread will get it. If all threads have the signal blocked, it's queued in the per-process queue. If there is no signal handler defined for the signal and the default action is to terminate the process with or without dumping the core the whole process is terminated.

Signal Catching Functions should be Reentrant

A signal handler `handler()` is registered for a call back on a signal occurrence. Now assume that this `handler()` was already in execution when the signal occurred. Since this function is call back for this signal so the current execution on this signal will be stopped by the scheduler and this function will be called again (due to signal).

If there are any global Variable which hold the values of the previous execution what happens ?

If there are any memory allocations what will happen ? `malloc()`

This is why Signal Handlers should be Reentrant

Pipes

Pipes are used to “collect” the output of one process and feed it as input to another process

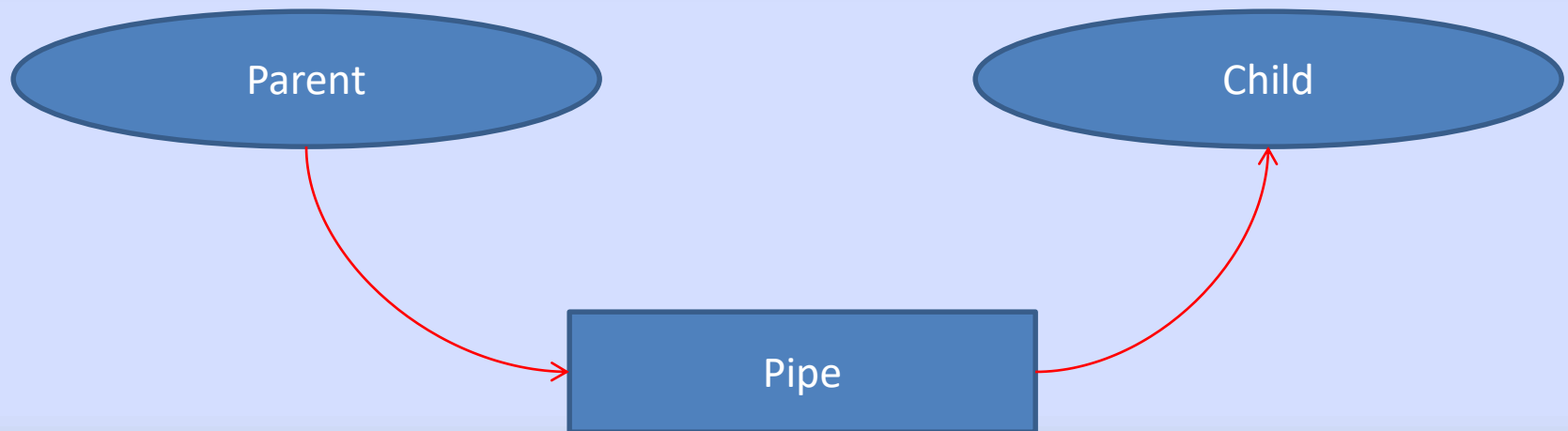
Eg: Searching for a file which has “hello” in its name , a local folder

```
ls -l | grep "hello"
```



Nature of Pipes

- Pipe are Half Duplex data flows only in one direction
- Pipe can be used only between processes that have a common ancestor



```

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

int main(void)
{
    int    fd[2], nbytes;
    pid_t  childpid;
    char   string[] = "Data: 9d234ffcc44 \n";
    char   readbuffer[80];

    pipe(fd);

    childpid = fork();

    if(childpid == -1)
    {
        perror("fork");
        exit(1);
    }

```

```

if(childpid == 0)
{
    /* Child process closes up input side of pipe */
    close(fd[0]);

    /* Send "string" through the output side of pipe */
    printf(" Child Process: %s \n",string);
    write(fd[1], string, (strlen(string)+1));
    exit(0);
}
else
{
    /* Parent process closes up output side of pipe */
    close(fd[1]);

    /* Read in a string from the pipe */
    nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
    printf("Parent Process: Received string: %s",
        readbuffer);
}
return(0);
}

```

If the parent wants to receive data from the child, it should close fd1, and the child should close fd0. If the parent wants to send data to the child, it should close fd0, and the child should close fd1. Since descriptors are shared between the parent and child, we should always be sure to close the end of pipe we aren't concerned with.

Warm-up Assignment

We have standard linux utilities ls , grep, wc etc

Write a C program that collect the output of one of these and pipe it as input into another – Do not use standard | operator

Eg:

To find the number of .jpg files in a Linux folder we use the following

```
% ls -l *.jpg | wc -l
```

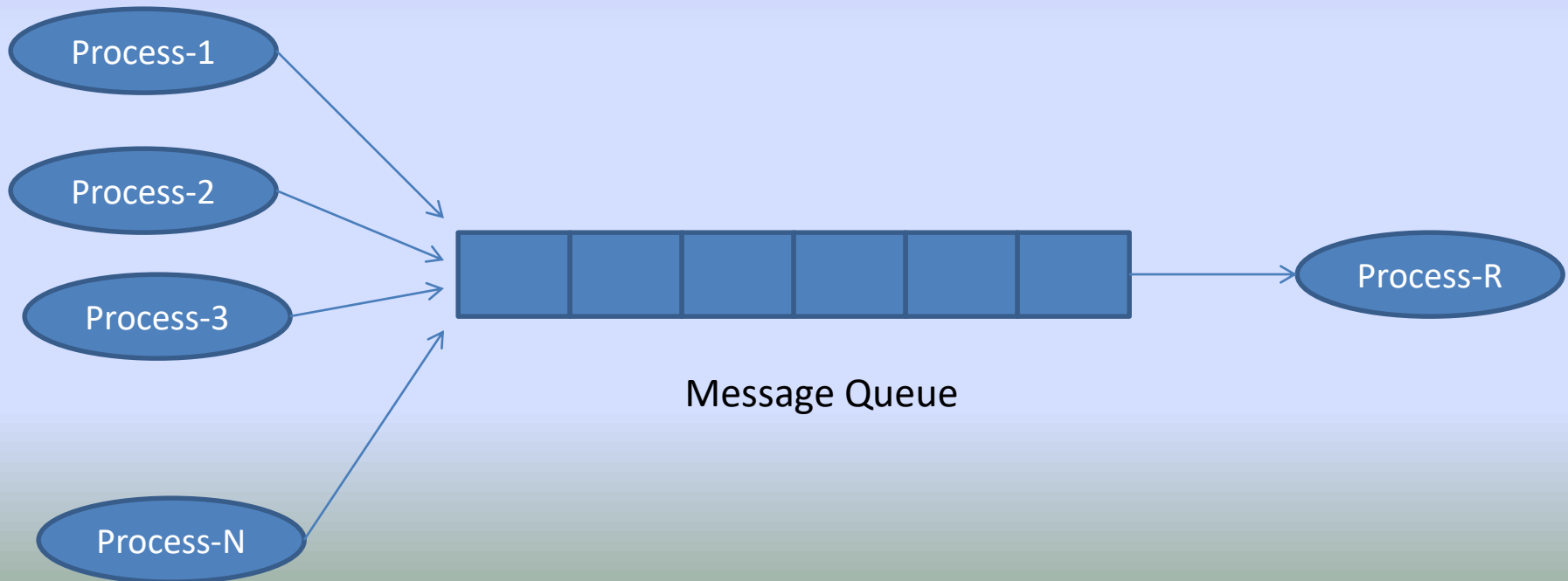
Your Job is to:

Run ls -l through a C- program and get the output and similarly run wc utility. “pipe” the output of ls -l to wc utility through your C program itself

Or search for a particular file name in output of ls -l using grep

Message Queues

Message queues allow one or more processes to write messages that will be read by one or more reading processes.




```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct my_msgbuf {
    long mtype;
    char mtext[200];
};

int main(void)
{
    struct my_msgbuf buf;
    int msqid;
    key_t key;

    if ((key = ftok("kirk.c", 'B')) == -1)
    {
        perror("ftok");
        exit(1);
    }

```

```

    if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1)
    {
        perror("msgget");
        exit(1);
    }

    printf("Enter lines of text, ^D to quit:\n");

    buf.mtype = 1; /* we don't really care in this case */

    while(fgets(buf.mtext, sizeof buf.mtext, stdin) != NULL)
    {
        int len = strlen(buf.mtext);

        /* ditch newline at end, if it exists */
        if (buf.mtext[len-1] == '\n') buf.mtext[len-1] = '\0';

        if (msgsnd(msqid, &buf, len+1, 0) == -1) /* +1 for '\0' */
            perror("msgsnd");
    }

    if (msgctl(msqid, IPC_RMID, NULL) == -1)
    {
        perror("msgctl");
        exit(1);
    }

    return 0;
}

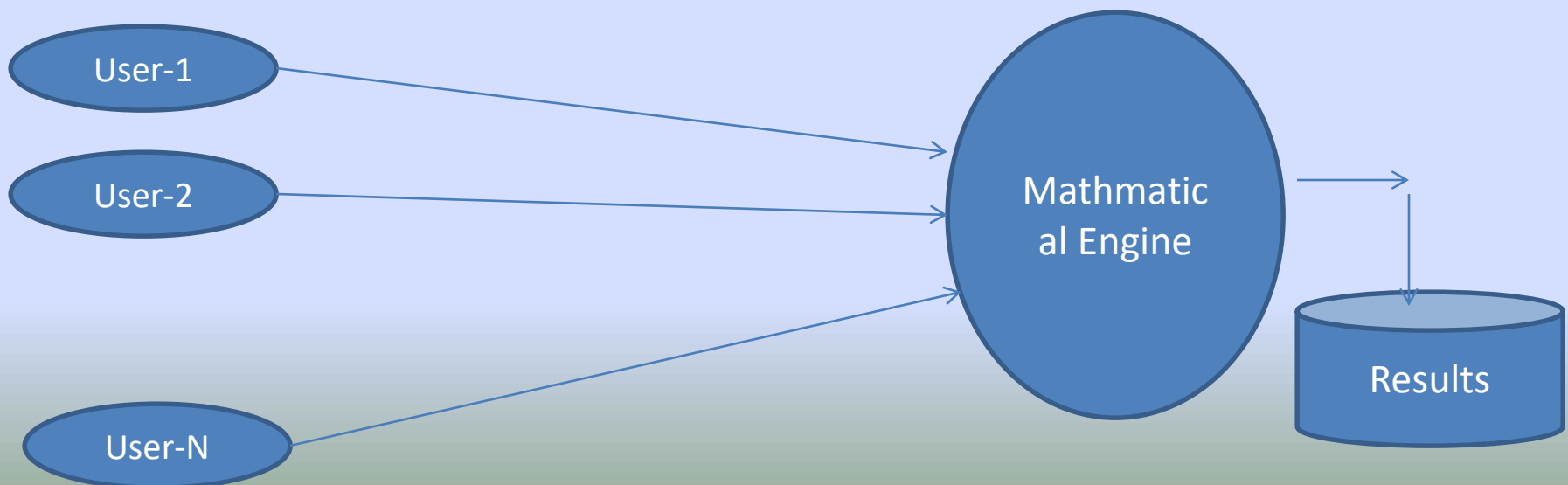
```

Warm-up Assignment-2

Software engineers are designing an application that runs on a Linux machine. The Core of the app is mathematical engine which processes complex mathematical equations based on Linear Algebra.

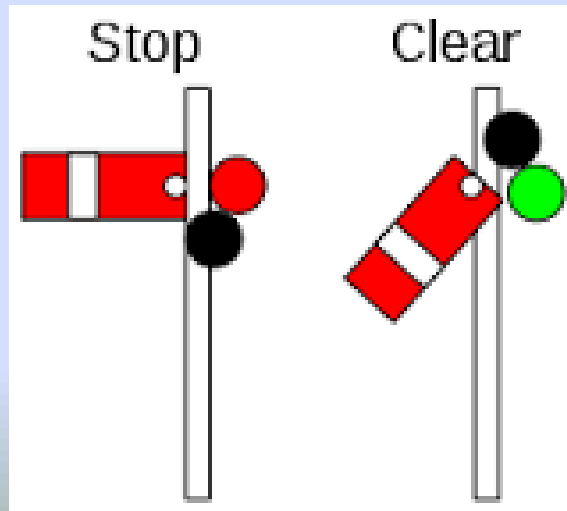
Multiple users would like to run this application through a client.

Design a client and server model to implement this app



Semaphore

Semaphores are a programming construct designed by E. W. Dijkstra in the late 1960s. Dijkstra's model was the operation of railroads: consider a stretch of railroad in which there is a single track over which only one train at a time is allowed. Guarding this track is a semaphore. A train must wait before entering the single track until the semaphore is in a state that permits travel. When the train enters the track, the semaphore changes state to prevent other trains from entering the track. A train that is leaving this section of track must again change the state of the semaphore to allow another train to enter.



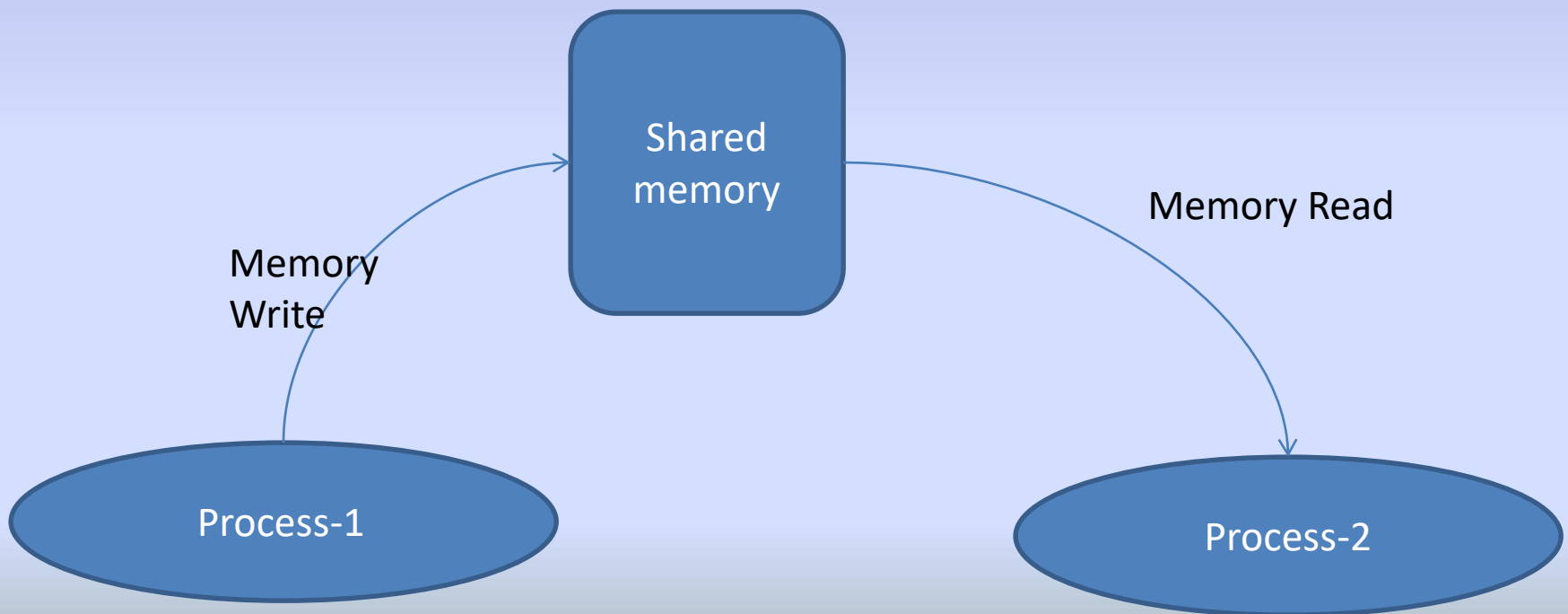
Semaphore API

No	API	Meaning
1	Ftok()	Generate a unique token.
2	Semget()	Create a new semaphore
3	semctl()	Once you have created your semaphore sets, you have to initialize them to a positive value to show that the resource is available to use
4	semop()	All operations that set, get, or test-n-set a semaphore use the semop() system call.
5	semctl(semid, 0, IPC_RMID);	Destroy a Semaphore

Shared Memory

Shared Memory allows two or more process to share a given region of memory. This is one of the fastest form of IPC because the data does not need to be copied Between process.

Large amount of data can be transferred between two process



Shared Memory API

No	API	Meaning
1	Ftok()	Generate a unique token.
2	Shmget()	Create a new shared memory or connect to an existing one
3	Shmat()	Attach the process to the memory
4	Shmdt()	Detach the process to the memory

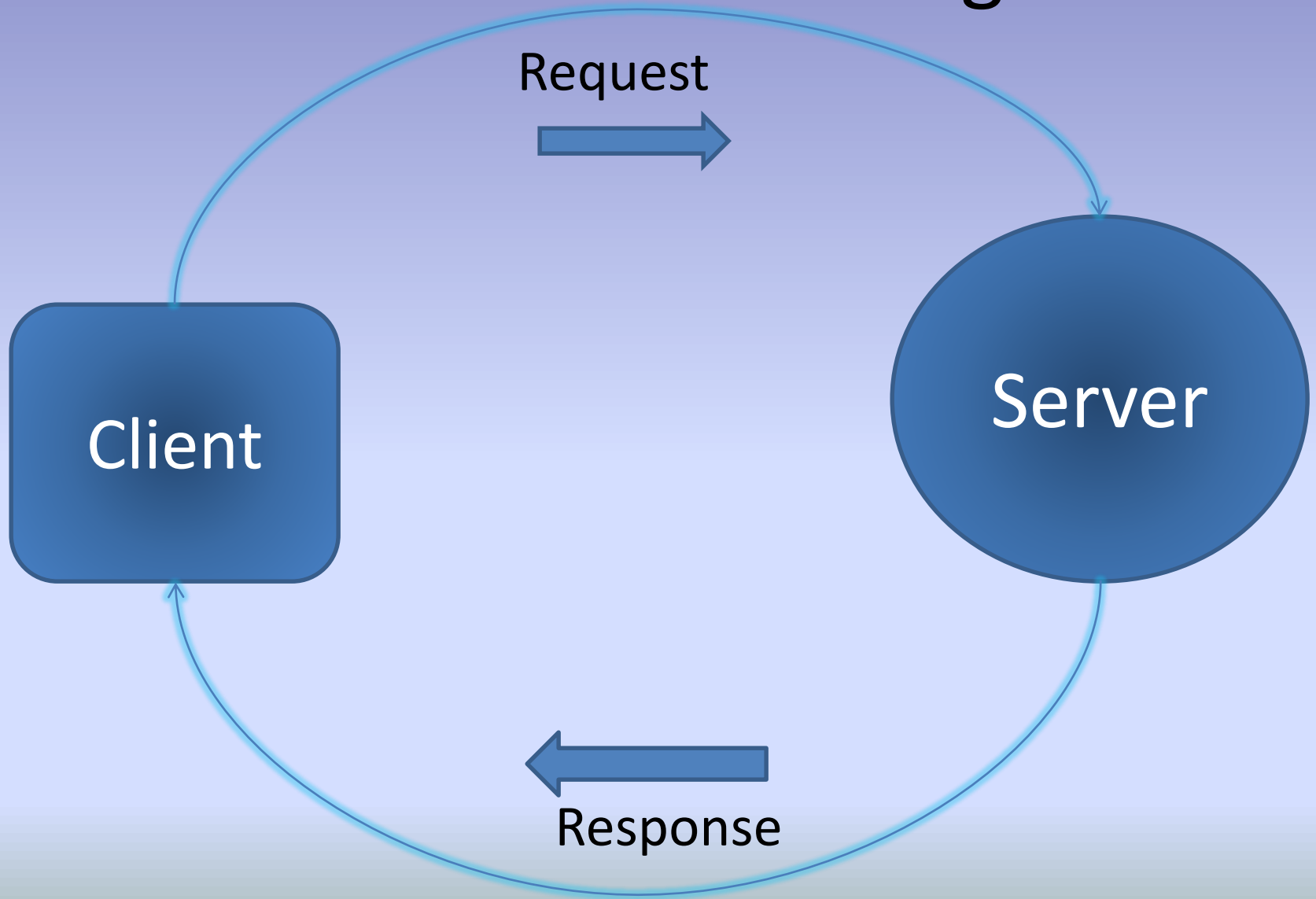
Warm-up Assignment-3

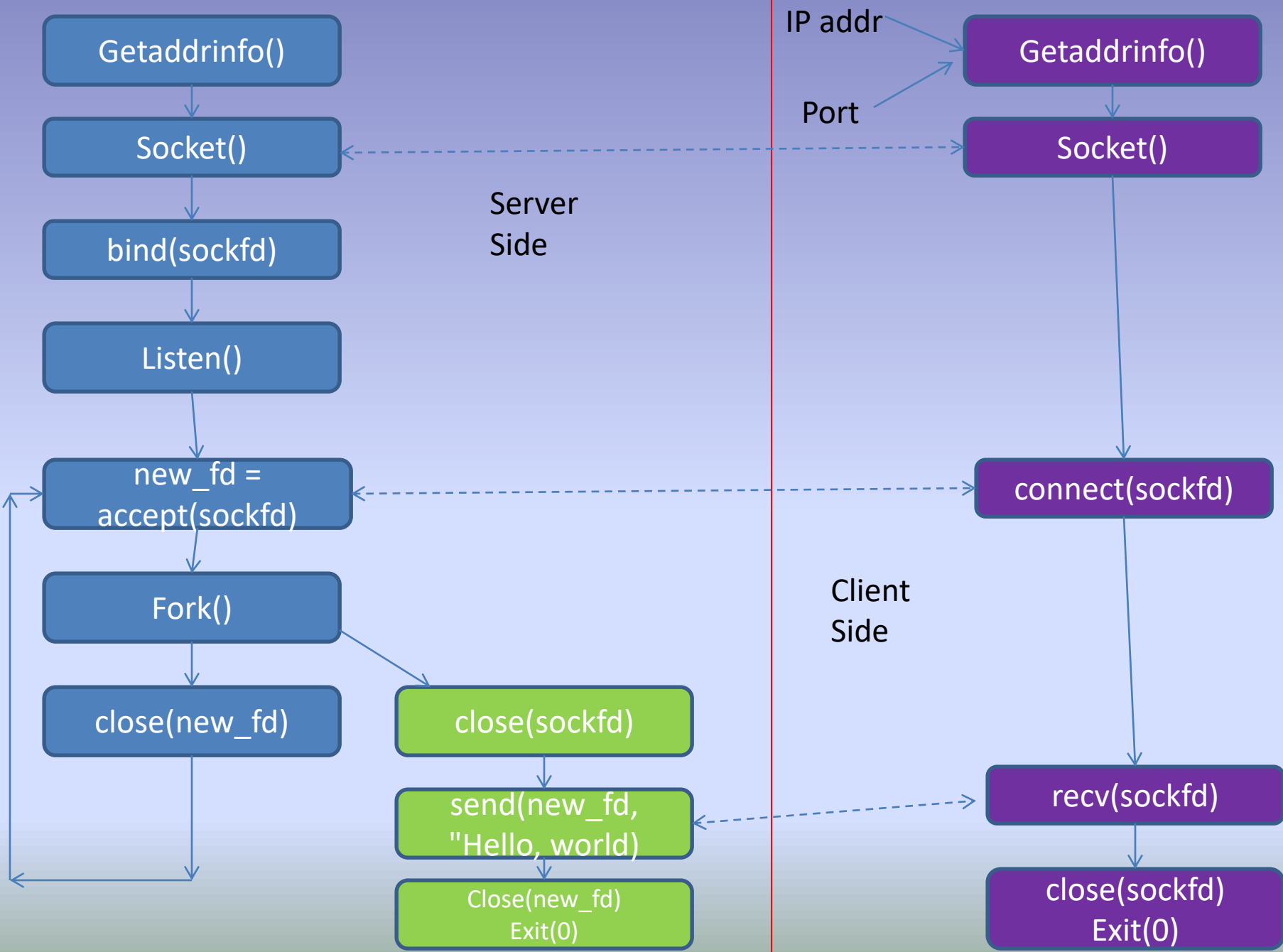
1. Re-write the sample program which uses Shared memory with parent and child created out of `fork()`
2. Implement Semaphore for the preventing accidental over write while reading in progress

Sockets

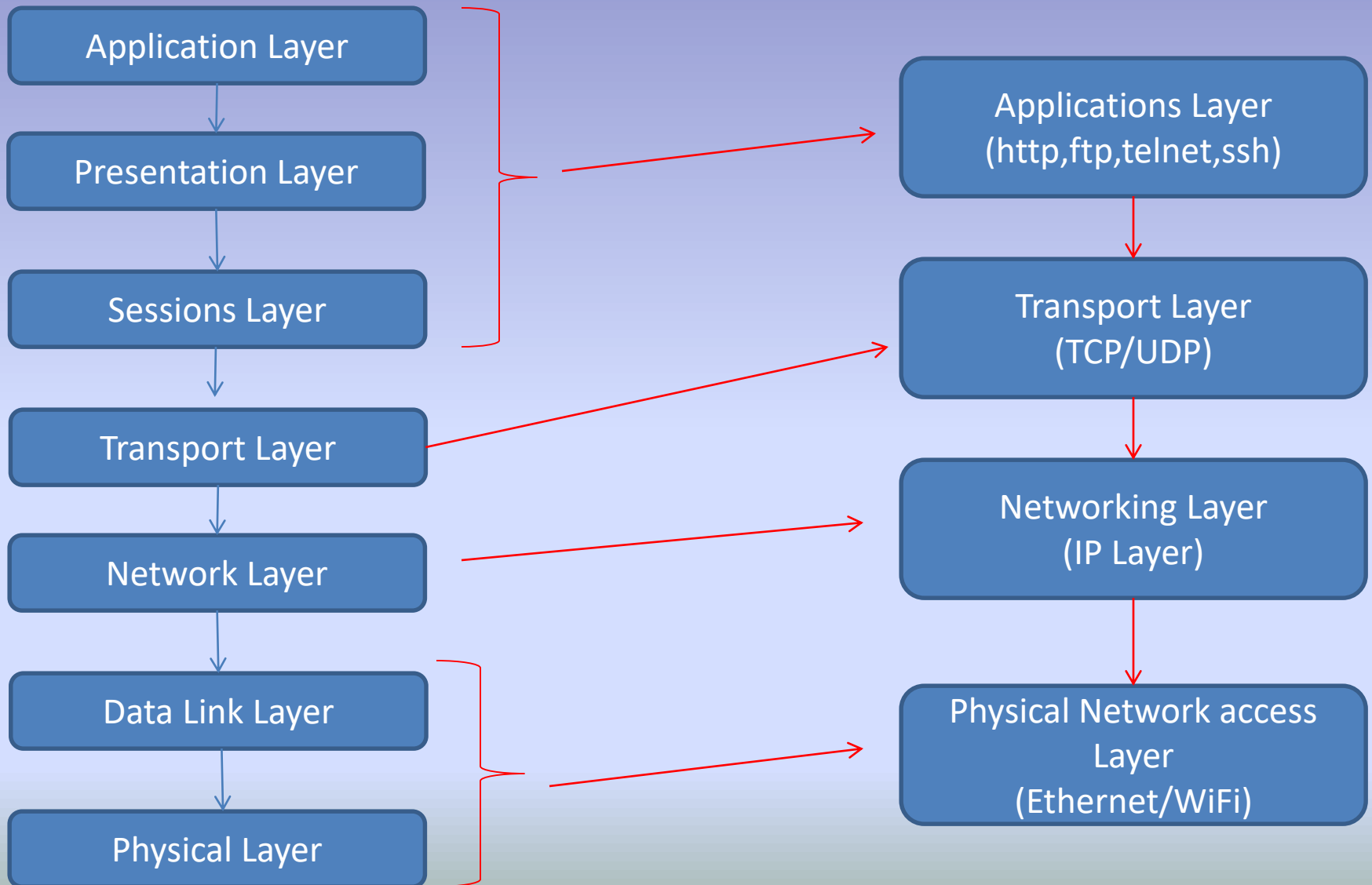
- Sockets are two way mechanisms for doing Inter Process communication
- All other IPC mechanisms makes use of Kernel internals where as Sockets are based on TCP/UDP/IP Protocols
- One of the most commonly used IPC in computing world today

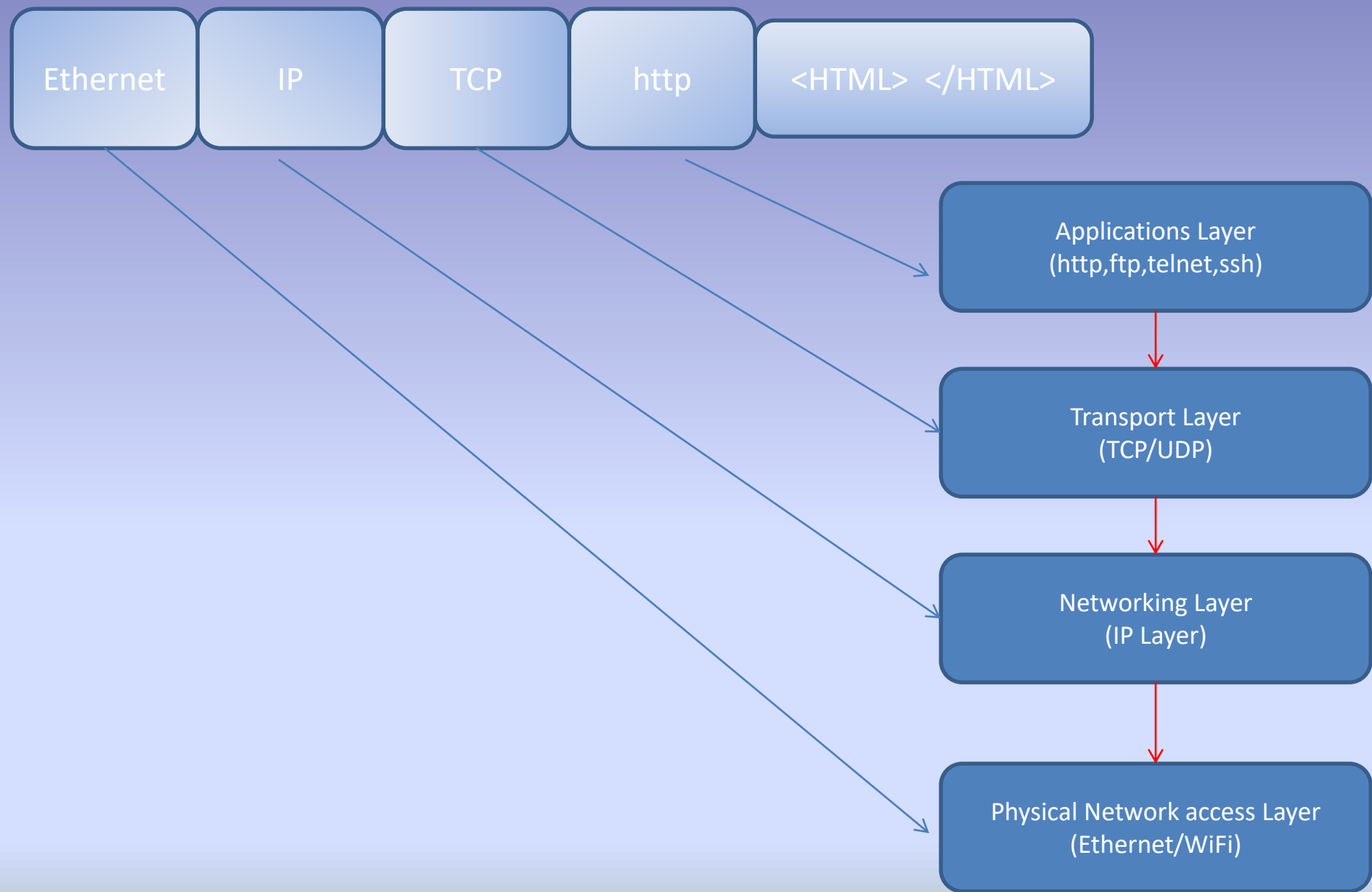
Client Server Paradigm





Implementation of OSI model





Data Encapsulation

IPC on VxWorks

No		
1	Binary Semaphores	it is fast and most general purpose semaphores.it is for synchronization and mutual exclusion.
2	Mutual Exclusion Semaphore	Mutual-exclusion semaphores offer convenient options suited for situations requiring mutually exclusive access to resources.
3	Counting Semaphore	Keep track of the number of times a semaphore is given, optimized for guarding multiple instances of a resource.
4	Signals	The signal facility provides a set of 31 distinct signals. A signal can be raised by calling kill().
5	Shared Memory	Shared memory objects provide high-speed synchronization and communication among tasks running on separate CPUs that have access to a common shared memory
6	Message Queues	Message queues allow a variable number of messages (varying in length) to be queued in first-in-first-out (FIFO) order Full duplex communication between two tasks generally requires two message queue, one for each direction.
7	Sockets	This library provides UNIX BSD compatible socket calls. Use these calls to open, close, read, and write sockets. These sockets can join processes on the same CPU or on different CPUs between which there is a network connection. The calling sequences of these routines are identical to their equivalents under UNIX BSD

I/O Operations

