COURSE: COMPUTER NETWORKS LABORATORY

**CSN-361**

REPORT ON

ASSIGNMENT 1

**MADE BY:**

**Name: Ghetia Siddharth**

**Enrollment Number: 17114033**

**CSE 3rd YEAR**

**Problem Statement 1**:

Write a C program in the UNIX system that creates two children and four grandchildren (two for each child). The program should then print the process-IDs of the two children, the four grandchildren and the parent in this order.

**Algorithms and data structures used in the implementation:**

Various **Data Structures** used are:

1. **In-Built integer data**:

* To store the integer value returned from the **fork()** function call so that we can identify whether the process running is child or parent process.

Various **Algorithms/Function Calls** used are:

1. **fork() :**

* System call fork() is used to create processes. It takes no arguments and returns a process ID. The purpose of fork() is to create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork():

1. If fork() returns a negative value, the creation of a child process was unsuccessful.
2. fork() returns a zero to the newly created child process.

3. fork() returns a positive value, the process ID of the child process, to the parent.

1. **getpid() and getppid() :**

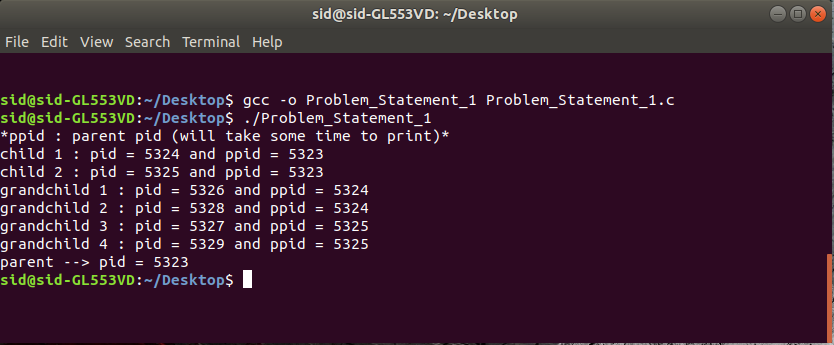
* getpid() returns the pid(Process Identifier) of the process.
* getppid() returns the Parent Process’s pid.

**Algorithm:**

1. Create a child process.
2. Within the child process create 2 child processes by calling fork() in appropriate manner as done for the parent process.
3. Now as the control returns to the parent process create another child process.
4. Within this child process create 2 child processes by calling fork() in appropriate manner as done for the parent process.

To print the pid in given order we need to place sleep() call function at various places for ordered execution of child and grandchild processes this is also called as **busy waiting**.

**Snapshots:**

****

**Problem Statement 2**:

Write a C++ program to print the MAC address of your computer.

**Algorithms and data structures used in the implementation:**

Various **Data Structures** used are:

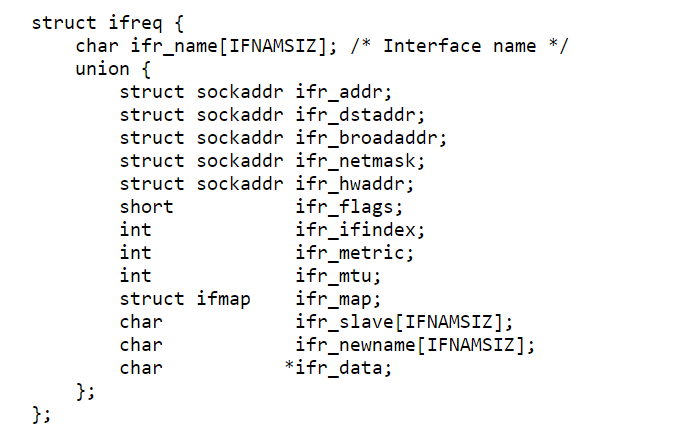
1. **In-Built integer data**:

* To store the integer value returned from the **socket()** function call so that we can identify whether the socket has been created successfully and also the socket’s file descriptor.

1. **struct ifreq:**

Linux supports some standard ioctls (input output controls) to configure network devices.

They can be used on any socket's file descriptor regardless of the family or type. Most of them pass an ifreq structure:



Normally, the user specifies which device to affect by setting ifr\_name to the name of the interface (here esn3s0). All other members of the structure may share memory.

Various **Algorithms/Function Calls** used are:

1. **socket() :**

* int **socket(int domain, int type, int protocol);**

socket() creates an endpoint for communication and returns a file descriptor that refers to that endpoint. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

It takes 3 arguments:

1. domain: Specifies the communication domain and the protocols to be used.
2. type: Specifies the communication semantics and socket type.
3. protocol: Specifies a particular protocol to be used with socket.
4. **ioctl() :**

* **int ioctl(int** *fd***, unsigned long** *request***, ...);**

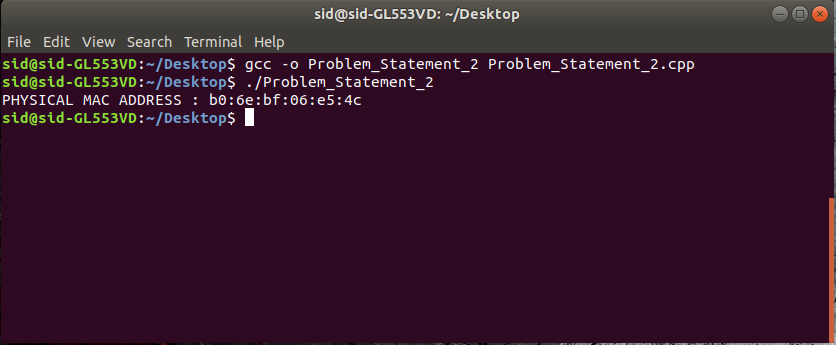
The ioctl() system call manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g., terminals) may be controlled with ioctl() requests. The argument fd must be an open file descriptor. The second argument is a device-dependent request code. The third argument is an untyped pointer to memory (here of struct ifreq\*) .

On success 0 is returned.

**Algorithm:**

1. Create a datagram socket.
2. Create a data of type struct ifreq (let it be mac).
3. Set the mac.ifr\_name attribute to enp3s0.
4. Call ioctl as ioctl(socket\_1, SIOCGIFHWADDR, &mac) and thus it will return the related information in mac data structure from which we can gather the MAC Address.

**Snapshots:**

****

**Problem Statement 3**:

Write your own version of ping program in C language.

**Algorithms and data structures used in the implementation:**

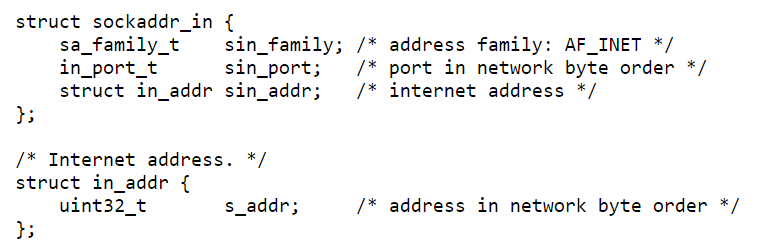
Various **Data Structures** used are:

1. **In-Built data structure**:

* **Integer:** To store the integer value returned from the **socket()** function call so that we can identify whether the socket has been created successfully and also the socket’s file descriptor and various other local uses.
* **char array, char\*, etc.** (for local use).

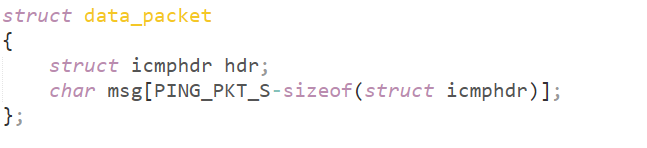
1. **struct sockaddr\_in:**

* struct sockaddr\_in is the structure used with IPv4 addresses (e.g. "192.0.2.10"). It holds an address family (AF\_INET), a port in sin\_port, and an IPv4 address in sin\_addr.



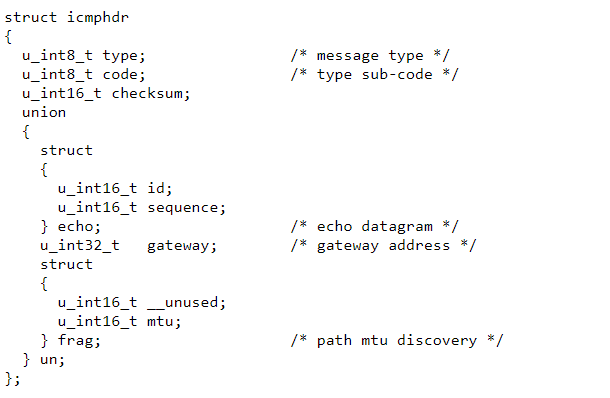
1. **struct data\_packet** (self defined):

To store the packet that is to be send and received again using icmp (Internet Control Message Protocol) protocol.



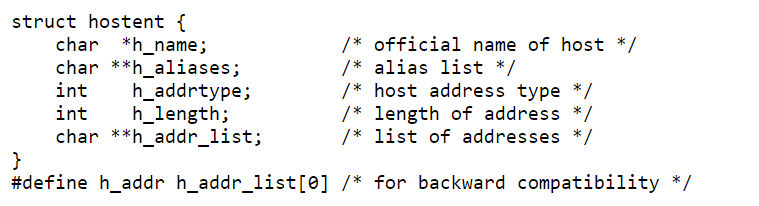
1. **struct icmphdr**:

To store the header information of packet that is to be send and received again using icmp (Internet Control Message Protocol) protocol.



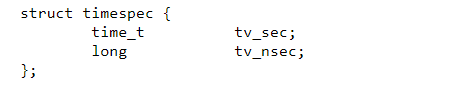
1. **struct hostent:**

It is used to represent an entry in the host database. It is defined as follows:



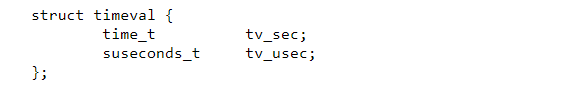
1. **struct timespec:**

Structure holding an interval broken down into seconds and nanoseconds.



1. **struct timeval:**

The struct timeval structure represents an elapsed time. It is declared in sys/time.h and is defined as follows:



1. and various other data structures.

Various **Algorithms/Function Calls** used are:

1. **socket() :**

* int **socket(int domain, int type, int protocol);**

socket() creates an endpoint for communication and returns a file descriptor that refers to that endpoint. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

It takes 3 arguments:

1. domain: Specifies the communication domain and the protocols to be used.
2. type: Specifies the communication semantics and socket type.
3. protocol: Specifies a particular protocol to be used with socket.
4. **ioctl() :**

* **int ioctl(int** *fd***, unsigned long** *request***, ...);**

The ioctl() system call manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g., terminals) may be controlled with ioctl() requests. The argument fd must be an open file descriptor. The second argument is a device-dependent request code. The third argument is an untyped pointer to memory (here of struct ifreq\*) .

On success 0 is returned.

1. **clock\_gettime() and clock\_settime():**

* **int clock\_gettime(clockid\_t clk\_id, struct timespec \*tp);**
* **int clock\_settime(clockid\_t clk\_id, const struct timespec \*tp);**

The functions clock\_gettime() and clock\_settime() retrieve and set the time of the specified clock clk\_id.

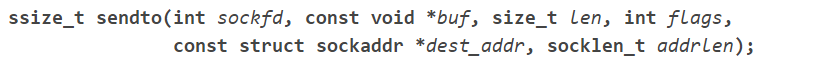
1. **setsockopt():**

* **int setsockopt(int socket, int level, int option\_name,**

**const void \*option\_value, socklen\_t option\_len);**

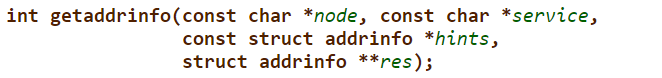
The setsockopt() function shall set the option specified by the option\_name argument, at the protocol level specified by the level argument, to the value pointed to by the option\_value argument for the socket associated with the file descriptor specified by the socket argument.

1. **sendto();**



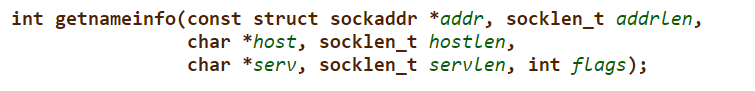
These functions send data to a socket. Generally speaking, send() is used for TCP SOCK\_STREAM connected sockets, and sendto() is used for UDP SOCK\_DGRAM unconnected datagram sockets. With the unconnected sockets, you must specify the destination of a packet each time you send one, and that's why the last parameters of sendto() define where the packet is going.

1. **getaddrinfo():**



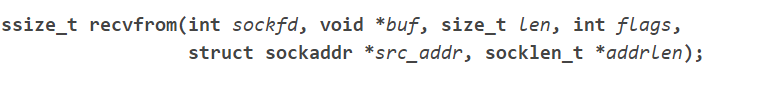
Given node and service, which identify an Internet host and a service, getaddrinfo() returns one or more addrinfo structures, each of which contains an Internet address that can be specified in a call to bind(2) or connect(2).

1. **getnameinfo()**:



The getnameinfo() function is the inverse of getaddrinfo(3): it converts a socket address to a corresponding host and service, in a protocol-independent manner.

1. **recvfrom():**



Once you have a socket up and connected, you can read incoming data from the remote side using the recv() (for TCP SOCK\_STREAM sockets) and recvfrom() (for UDP SOCK\_DGRAM sockets).

Both functions take the socket descriptor, a pointer to the buffer , the size (in bytes) of the buffer, and a set of flags that control how the functions work.

recvfrom() additionally takes the address from where to receive the data.

**(SELF DEFINED FUNCTIONS):**

1. **checksum():**



To generate the hash of the data and to verify that the data is sent and received without any loss.

1. **Interrupt\_func():**



To enable interrupting handling this is passed as an argument in signal() system call and tells what to do on signal interrupt.

1. **DNS\_lookup():**



To obtain the ip address and the socketaddr\_in type reference to the given host name/domain like [www.google.com](http://www.google.com)

1. **rev\_DNS\_lookup():**



To obtain the host name from the given ip address.

1. **ping\_testing()**



The function which will act as the ping program on receiving appropriate arguments like socket file descriptor, hostname, etc.

and others functions.

**Algorithm for ping program:**

**Implementation**

The steps followed by a simple ping program are:

1. Take a **hostname** as input
2. Do a **DNS lookup**

DNS lookup can be done using **gethostbyname()**. The gethostbyname() function converts a normal human readable website and returns a structure of type **hostent** which contains IP address in form of binary dot notation and also address type.

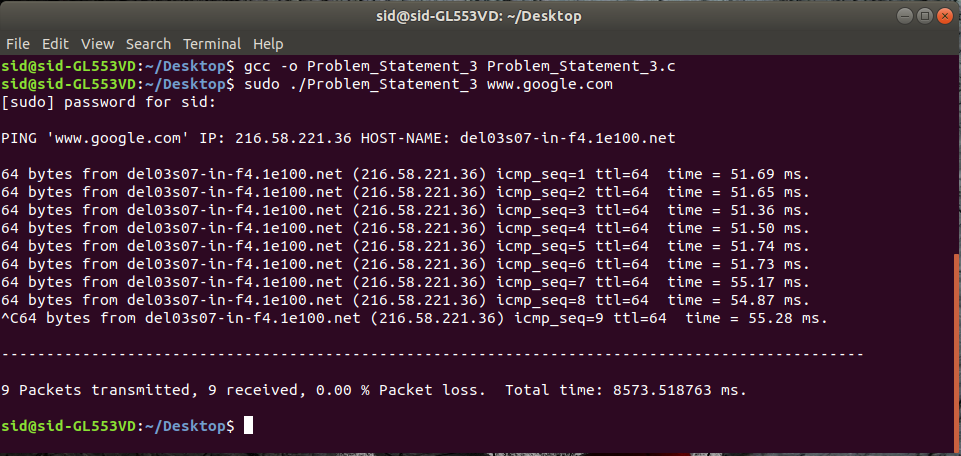
1. Some ping programs like the one given with ubuntu support **reverse DNS lookup**.  
   Reverse DNS lookup is performed using **getnameinfo()**, and it converts dot notation IP address to hostname.

for example, the pinging of google.com frequently gives a strange address:  
bom07s18-in-f14.1e100.net

This is as a result of a reverse DNS lookup.

1. Open a **Raw socket** using SOCK\_RAW with protocol as IPPROTO\_ICMP.  
   Note: raw socket requires super user rights so you have to run this code using sudo
2. When crtl + C is pressed, ping gives a report. This interrupt is caught by an interrupt handler which just sets our pinging looping condition to false.
3. Here comes the main ping sending loop.  
   We have to:
   1. Set the **ttl option** to a value in the socket  
      TTL value is set to limit the number of hops a packet can make.
   2. Set the **timeout** of the recv function  
      If timeout is not set, recv will wait forever, halting the loop.
   3. Fill up the **icmp packet**  
      As follows:
      1. Set packet header type to ICMP\_ECHO.
      2. Set id to pid of process
      3. Fill msg part randomly.
      4. Calculate checksum and fill it in checksum field.
   4. **Send** the packet
   5. Wait for it to be **received**.  
      The main problem here is that the packet received does not mean that that the destination is working.  
      Echo reply means destination is OK. Echo reply is sent from destination OS kernel.

**Snapshots:**

****

**Problem Statement 4**:

Write a C program to find the host name from IP address.

**Algorithms and data structures used in the implementation:**

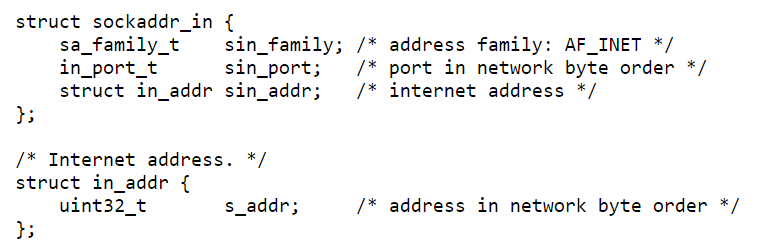
Various **Data Structures** used are:

1. **In-Built data types and structures**:

* **Integer:** To store the integer value returned from the **socket()** function call so that we can identify whether the socket has been created successfully and also the socket’s file descriptor.
* **Char \*, etc.**

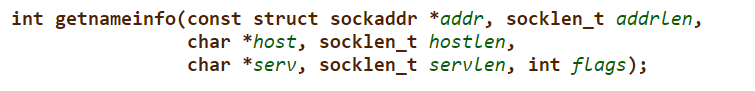
1. **struct sockaddr\_in:**

* struct sockaddr\_in is the structure used with IPv4 addresses (e.g. "192.0.2.10"). It holds an address family (AF\_INET), a port in sin\_port, and an IPv4 address in sin\_addr.



Various **Algorithms/Function Calls** used are:

1. **getnameinfo()**:



The getnameinfo() function is the inverse of getaddrinfo(): it converts a socket address to a corresponding host and service, in a protocol-independent manner and thus from this we can get the hostname from given ip address.

1. **inet\_addr():**

To convert the given ip address ( as char\* ) into specified format.

**Algorithm:**

* 1. Input the ip address.
  2. Create a data of type struct socketaddr\_in.
  3. Set its parameters like sin\_family to AF\_NET, sin\_addr.s\_addr to inet\_addr(IP), etc.
  4. Call getnameinfo() with appropriate arguments.
  5. As a result we will get the hostname of the given ip address.

**Snapshots:**

