Que 9, 4. How do you implement a concurrent server in UNIX? Explain.

Ans: Concurrent server are used when we want to handle multiple clients at the same time. The simplest way to write concurrent server in UNIX is to fork() a child process to handle each client.

```
pid_t    pid;
int      listenfd, connfd;

listenfd  = Socket(... );
     /* fill in sockaddr_in{} with server's well-known port */

Bind(listenfd, ... );
Listen(listenfd, LISTENQ);

for ( ; ; ) {
          connfd = Accept (listenfd, ... ); /* probably blocks */

          if( (pid = Fork()) == 0) {
                    Close(listenfd);  /*child closes listening socket */
                    doit(connfd);              /*process the request */
                    Close(connfd);             /*done with this client */
                    exit(0);                   /* child terminates */
          }

          Close(connfd);             /* parent closes connected socket */
}
```

When a connection is established, accept returns, the server calls fork, and the child process services the client (on connfd, the connected socket) and the parent process waits for another connection (on listenfd, the listening socket). The parent closes the connected socket since the child handles the new client.

Visualizing the socket and connection of concurrent server

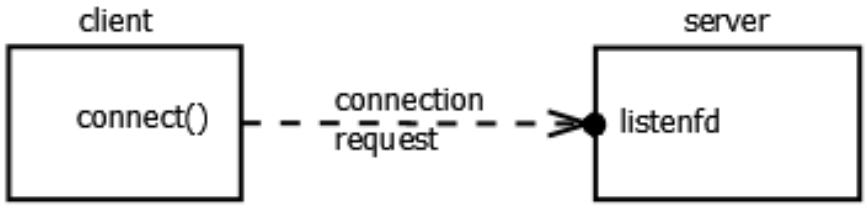

Figure 1. Status of client/server before call to accept returns.

Figure 1 shows the status of the client and server while the server is blocked in the call to accept and the connection arrives from the client. Immediately after accept returns, we have the scenario shown in Figure 2. The connection is accepted by the kernel and a new socket, connfd, is created. This is a connected socket and data can now be read and written across the connection.

Figure 2. Status of client/server after return from accept.

The next step in the concurrent server is to call fork. Figure 3 shows the status after fork returns.

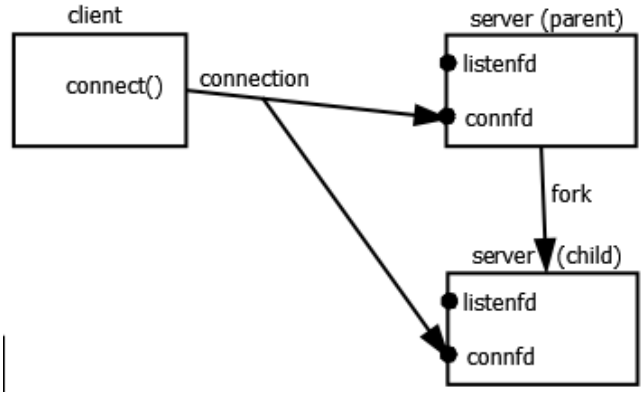

Figure 3. Status of client/server after fork returns.

Notice that both descriptors, listenfd and connfd, are shared (duplicated) between the parent and child.
The next step is for the parent to close the connected socket and the child to lose the listening socket. This is shown in Figure 4.

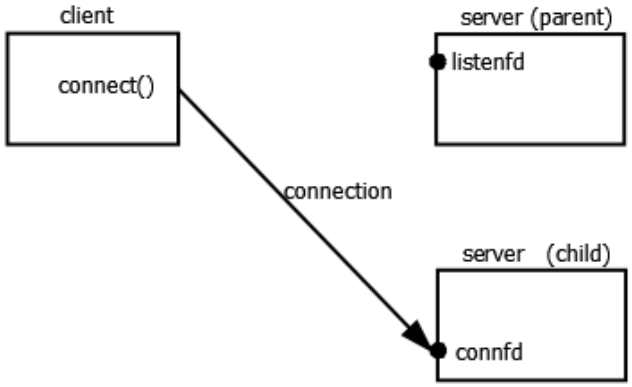

Figure 4. Status of client/server after parent and child close appropriate sockets.

This is the desired final state of the sockets. The child is handling the connection with the client and the parent can call accept again on the listening socket, to handle the next client connection.

Que 10, 5. Compare and contrast Synchronous vs Asynchronous I/O modes in UNIX programming?

Ans:

Synchronous	Asynchronous
A *synchronous I/O operation* causes the requesting process to be blocked until that I/O operation completes.	An *asynchronous I/O operation* does not cause the requesting process to be blocked.
It includes four I/O models: blocking, nonblocking, I/O multiplexing, and signal-driven I/O.	It includes only asynchronous I/O.
The actual I/O operation (recvfrom) block the other process.	Multiple operations run concurrently.
Multiple phase is handled differently for each model.	Multiple phases handled in same way.
Blocking is done to copy data from kernel to user.	Waiting and copying goes concurrently.

Que 11, 6. Compare and contrast I/O Multiplexing and signal driven I/O.

Ans:

I/O Multiplexing	Signal driven I/O
It is synchronous mode which uses select or poll function to block the system calls instead of blocking the actual I/O system call.	It is synchrounous mode that uses SIGIO signal to notify about the descriptors.
Select() blocks the call and waits for datagram, socket to be readable. This blocks the call.	Sigaction() system call is used to install signal handler to return datagram, but the call is not blocked.
When select returns, recvfrom() is used to copy the datagram into application buffer.	When the datagram is ready, the SIGIO signal is generated for the process.

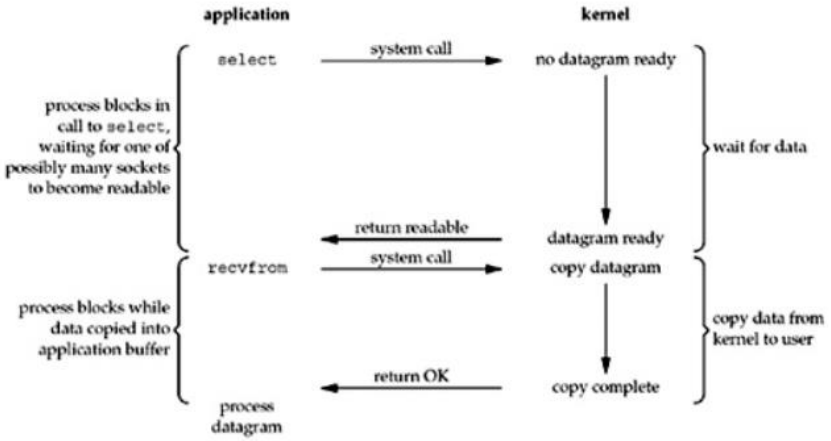

Figure 5. I/O Multiplexing

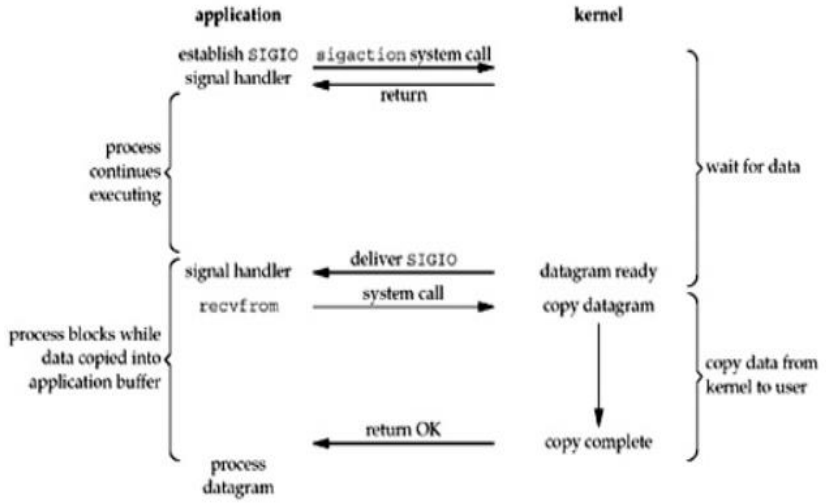

Figure 6. Signal Driven I/O

Que 12, 7. Write a simple program to resolve host name to an IP address.

Ans:

```
#include <winsock2.h>
#include <Windows.h>
#include <stdio.h>

//#pragma comment (lib, "ws2_32.lib")

int main(void){
	WSADATA wsaData = { 0, };
	struct in_addr addr = { 0, };
	struct hostent * res;
	int i = 0;

	WSAStartup(MAKEWORD(2, 2), &wsaData);

	res = gethostbyname("localhost");
	while (res->h_addr_list[i] != 0)
	{
		addr.s_addr = *(u_long *)res->h_addr_list[i++];
		printf("IP Address: %s\n", inet_ntoa(addr));
	}

	WSACleanup();
}
```

Que 13, 8. Write a simple program to resolve service name to corresponding port.

Ans:

```
#include <stdio.h>
#include <winsock2.h>

int main() {
	WSADATA wsa;
	WSAStartup(MAKEWORD(2, 2), &wsa);

	struct servent *var = getservbyname("aa", "tcp");
	if(var == NULL) {
		char msgbuffer[100];
		printf("Error: %d\n", WSAGetLastError());
		int error_code = WSAGetLastError();

	FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM,NULL,error_code,MAKELANGID(LANG_NEUTR
AL,SUBLANG_DEFAULT),msgbuffer,sizeof msgbuffer,NULL);
		printf("%s\n",msgbuffer);
	}
	else {
		printf("Name: %s Port:%u Protocol:%s\n", var->s_name, var->s_port, var->s_proto);
	}
	WSACleanup();
	return 0;
}
```