Client Server Computing (Elective - II)

Dept. Computer Science, SJCET, Palai
MODULE 1

CLIENT SERVER COMPUTING

Introduction

Network communication is mainly based upon the interaction between the client and server. Heterogeneous computing is the term used for the description of the client server interaction. Heterogeneous computing is used in the divorce computing environment.

There are many issues regarding the heterogeneous computing that face both programmers and MIS personnel. The three main components of any system are its hardware, software programs and operating system. The hardware platforms must be able to support multiple operating system, networks and protocols.

The OS must deal with logical connections not only between their peer components, but with other OS. Application developers make code work over diverse hardware, OS and networks.

HETEROGENEOUS COMPUTING

Heterogeneous computing means the ability to communicate with other dissimilar OS and protocols. Heterogeneous computing is the term used for describe this diverse computing environment. Each application vendor must strive to support key platform for its target software consumer. Developers are faced with the dilemma of how to make application software port from platform to platform with as little difficulty, expense, and problems as possible.

CROSS-PLATFORM COMPUTING

It is defined as the implementation of technologies across heterogeneous environments. When creating cross platform applications, developers are saddled with many issues. Providing the following capabilities is imperative for the application developer:
The application must run on multiple platforms.
1. It must have the same look and feel as on other platforms.
2. It must integrate with the native operating environment.
3. It must behave the same on all platforms.
4. It must be maintained simply and consistently.

Running on multiple platforms:

The application must be deployed on many computer systems. A Proliferation of computer hardware and OS makes up the current computing climate. Applications must support these environments to give users freedom of choice for deployment.

Maintaining the same look and feel on all platforms:

The application should maintain the same look and feel on all platforms. This is very crucial, as application users should not be unfamiliar with a product because it is implemented on different systems. Each OS system platform has similar graphical interface capabilities.

Supporting the native operating environment:

Application must utilize native features of the target platform. By supporting the native environment, the developer does not force in additional constraints or software on the user.

Behaving the same on all platforms:

The software must behave similarly on all systems. It is very important for applications to provide consistency to users.
Simple and consistent maintenance:

Developers should ease the integration of software into new environments by providing similar maintenance of application programs. Installation and update facilities vary from platform to platform.

DISTRIBUTED COMPUTING

Distributing computing is technology architecture with wonderfully promising benefits and advantages. Distributing computing involves the distribution of work among more than one machine. Distributing computing is broader in that many machines may be processing the work on behalf of client machines.

Advantages:

- Execution is very fast.
- Reduce the network traffic.
- Reduce the usage of memory.

COST OF CLIENT SERVER COMPUTING

For implementing the client server technology, a substantial cost is required. The cost includes system support, maintenance, and training, labor costs. Labor costs compromise almost half of the total costs of a client server information system. Since there are tremendous benefits and advantages to the technology, it cannot implement economically feasible.

Implementing a client server system will prove to be a challenging yet rewarding process. Knowledge, experience, initial costs, and exposure to client server computing will provide a tremendous base for leveraging subsequent project. Initial exposures however will probably not yield a bottom line savings or make IS or application developers more productive.
1. SYSTEM SUPPORT AND MAINTENANCE

Implementing a client server system will prove to be a challenging yet rewarding process. Knowledge, experience, initial costs, and exposure to client server computing will provide a tremendous base for leveraging subsequent project. Initial exposures however will probably not yield a bottom line savings or make IS or application developers more productive.

2. HARDWARE AND INFRASTRUCTURE COSTS

The infrastructure costs of client server computing were less than those of a comparative minicomputer based solution. The difference lies in terms of networks, gateways, bridges and routers needed to be purchased for the initial installation. PC’s are generally very low in cost and are always getting cheaper. In fact, this downward price trend, which is sure to be maintained into the future, will push the overall infrastructure costs of client server computing down even further. In addition, the component costs in microprocessor based environments are
sure to decrease as well, effectively lowering the costs of OS for the client, the server and for associated peripheral hardware.

3. SOFTWARE DEVELOPMENT COST

One of the greatest promises of client server computing is its development technology, methodologies, and tools. Comparative application development costs were 29% less expensive for client server implementations than for the minicomputer technologies. The cost difference is due to many factors including the development tools available and the inherent comparative ease of developing applications using those tools.

4. LABOR COSTS: Training, Administration, Support

The labor costs attributed to a client server technology deployment are numerous and high. As with any new technology, the initial and ongoing training of programmers, IS staff and users is to be expected. Total labor costs are 34% less for a minicomputer system deployment than they are for a client server one, according to Forrester. Labor costs themselves account almost 50% of the total cost of a client server implementation. Skilled labor comes at a high premium, especially in the initial phase of the new technology.

The ongoing maintenance and support of client server networks is a major cost factor facing cooperation’s today. In fact, management costs are magnified with client server because of the inherent manageability problems associated with distributed networks. Traditional main frame and mini-computers provide or a central management strategy with a central machine whose tools are time tested and proven. Client server management tools are just beginning to become robust enough to support large, disperse deployments. Managing distributed networks is a difficult task made worse by heterogeneous networks. Management solutions must be far reaching and well integrated for client-server technology to proliferate.
WAYS FOR PRESERVING EXISTING INVESTMENTS:

Economic constraints very often force companies to limit their spending on new computer resources and fully utilize existing ones. Weighing user’s productivity on outdated systems against the cost of replacing those systems is a delicate balancing act. Client-server promises both to preserve existing investments in computers and ease the pain of transition to newer systems. To do this, client hardware requirements are reduced and expensive server machines and peripherals are effectively shared. Client-server computing may benefit the bottom line of cooperation’s.

➢ Reducing client hardware requirements:

Client server architecture, which is designed to distribute functionality between two or more machines, reduces the client’s hardware requirements. The introduction of the server place a significant role by decreasing the amount of client processing. Resource utilization is thus reduced at these client machines and, as a result, less powerful existing machines may have their life extended.

➢ Reductions in client processing:

Many applications that use client-server technology spent a significant amount of execution time idling and waiting for the server. A notable change in client processing power makes only a slight improvement in client execution times. The client in effect, has become less dependent on machine capabilities when off-loading work to the server computers.

The significant reduction in client processing can be used in one of the two ways:
- Systems with extra CPU cycles can be used to run other programs or process other work at the client computer.
- Systems not capable of running multitasking OS can be preserved. These systems might not be able to participate in some areas without the use of client-server, as they aren’t capable of the required amount of work in a client-only solution.

<table>
<thead>
<tr>
<th>COST COMPONENT</th>
<th>CLIENT SERVER ( % )</th>
<th>MINICOMPUTER ( % )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Development</td>
<td>28</td>
<td>40</td>
</tr>
<tr>
<td>Infrastructure</td>
<td>25</td>
<td>27</td>
</tr>
<tr>
<td>System Support</td>
<td>40</td>
<td>30</td>
</tr>
<tr>
<td>Training</td>
<td>7</td>
<td>3</td>
</tr>
</tbody>
</table>

Table: Client Server Vs Minicomputer

**CLIENT SERVER COMPUTING**

Client server technology and the accompanying tools for the development provide the fundamental building blocks for high performance, highly leveraged and highly accessible information systems. Systems generated with client server computing have the potential to increase the productivity of end users. End users will have broader access to data and the ability to model it quickly.
BENEFITS OF CLIENT SERVER COMPUTING

- Reduced total execution time.
- Reduced client CPU usage.
- Increased client CPU usage.
- Reduced client memory usage.
- Reduced network traffic.

Reduced Total Execution Time:

Applications deployed as client only applications incur all processing on one CPU. The total execution time is sure to decrease when multiple machines are co-operating on the same program. If the server application becomes backlogged as it processes request from many clients, total execution time may actually increased. Thus it is very important for the application developer to develop scalable client server applications that avoid over burdening the server.

Reduced Client Server Usage:

By offloading work to a server computer the workstation machine may see a reduction in the amount of CPU usage. There are many advantages to reducing client CPU usage. The first is client application may take advantages of the additional CPU cycles that become available. The second is the application developer may determine what mix of client and server code execution best suits the particular application and the system platform.

Increased Client Server Usage:
Distributed computing maximizes the efficiency of the network by using the available CPU cycles at client stations. In such a scenario, these would be thought of as distributed work nodes. Client computers have the capacity to process information cooperatively should be utilized for that purpose. Client server efficiency is maximized in such an environment because CPU cycles at the client as well as the server are leveraged to create a more effective overall system. In order for clients operate in such a scenario; they probably need to use an advanced OS in order to provide multitasking and other capabilities. Clients that consist of such products and have enough horsepower will work very well to assist in cooperative processing.

**Reduced Client Memory Usage:**

Client applications employing client server technology may often gain additional advantages with client server computing. Memory usage may be decreased when implementing client server, because many pertinent features are to be deployed on another system. Another code as well as third party modules may be migrated to other platforms, thus benefiting the users of these systems. Memory constraints are very important to many users, so easing the memory crunch is much appreciated.

When third party functions and features used by the applications are also migrated to the server platform, an inherent reduction in memory and processing usage is obtained.

**LOCAL DATABASE**

- **Client Application**
- **Reporting Functions**
- **Database Interface**
- **Database**
CLIENT SERVER DATABASE

Fig: A client reporting application with a local and client server database.

Reduced Network Traffic:

Another important benefit of client server computing is the reduced network traffic. The client server model is designed for the client to initiate a request and for the server process
and respond to the request. With this model the actual data manipulation transpires on the server computer and is usually not transferred to the client station during processing. The client computer only receives data when the unit of work is completed. As a result fewer network transmissions are required to compute task.

Reduced network traffic may provide extremely significant benefit to the end user on a heavily used network. Network hardware systems have become faster, new topologies have emerged, bridging, and channel splitting have offered significant relief of network traffic. Anything that relieves the constraint of network resources is welcome, and the client-server model of computing may provide additional relief to over utilized network systems.

THE DIVISION OF LABOR BETWEEN CLIENT AND SERVER

The client server communication is with the help of much application as logic components. Many application components will execute more efficiently on either the client or server. For instance, logic components that may be shared among the clients might be placed at server for efficiently. If one server is processing the shared logic, the client will be freed to process other application components. In addition, if many clients are active in the processing of the non centralized application logic, then the overall system becomes more scalable.

The efficient execution components for the client server communication are:

Client:-
- Presentation/Display
- User Interaction
- Application Logic
- Request Formulation

Server:-
- Queries against shared resources
- Management: Application and data
- Transaction processing
- Centralized application logic
- Communications
- Computations

**CLIENT SERVER IMPLEMENTATION**

One of the keys to implementing successful client server applications is the proper distribution of functionality and programs between the client and the server machines. Choosing this application code mix is one of the most important architectural challenges to client server development. The correct distribution of code will produce the most optimal execution for both client and server. If either machine becomes overloaded, the benefits of the technology are reduced.

**MODULE-2**

**Fundamentals of Client-Server Design**

Client server systems are the coordination of involvement between two logical systems client and server and their application logic components. Many logic components will execute most efficiently on the client many will execute most efficiently on the server. It is
therefore not important to qualify which application components should execute where. Where components should execute is strictly a design issue for client server developers and should not be part of a formalized definition.

Traditional application development mandates that programmers give attention to application logic and its presentation to the user. With client server this doesn’t change. Managing this logical component is the essence of client server computing. Depending on the deployment of the client server application, the user interface issues may or may not change. For instance a downsized mainframe application might need to redesign a front end to deal with graphical user interface issues for the first time.

Applications already hosted on GUI platforms however probably won’t be impacted any differently with user interface issues.

- Request-Response component architecture
  Designing applications to become client server requires proper architecture. Because portions of an application execute on more than one system, client server software must be modularized for distribution among different machines. Separating application functionality into components makes writing client server applications much easier because components provide well-defined locations where application may become distributed.

  Conceptually application logic should be separated into independent components, where a requesting component makes requests and expects results to be returned from a responding component. This logical separation better prepares applications for distribution across a network.

- Implementing Client and Server system interaction
  The interaction between client and server systems may be implemented with a variety of techniques. The majority of the new code required for client server application development will consist of the coordination of client and server components. Traditionally applications make requests from components.
through a procedural interface. If however the procedure to be called is located on a different machine, the standard call and return mechanism will need to be altered.

In addition many of today’s popular client operating systems are event driven and message based in nature. These systems and the coordination of parts requires a different interface for distribution between client and server, There are many ways to manage these interactions. Application work, distributing functionality and managing the communications between client and server must be divided logically.

**Managing the interaction of client and server**

The distribution of application components across a network means that some means of interaction must be established. This usually involves new skill requirements for developers. Fortunately many new techniques and solutions are available today to ease the burden for application programmer.

There are three layers to be defined for application components to interact an application interaction protocol, a communication technique, and network protocol.

Application interaction protocols:

<table>
<thead>
<tr>
<th>Client Server application interaction protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RPC</strong></td>
</tr>
<tr>
<td>IPX/SPX</td>
</tr>
</tbody>
</table>
Layers of component interaction between client and server

Application interaction protocols are uniquely defined for each program and are used to logically describe the data being passed between client and server components. This level of protocol is usually chosen in conjunction with the communications technique but is independent of the underlying network protocol used to actually transfer the data.

Communication techniques

Communication techniques are well defined which give the developer the programmatic mechanism to transfer application data from client to server.

Network Protocols

There are a wide variety of network protocols. Client server developers were required to choose a communications protocols and then code by hand the interaction of client and server.

Middleware to ease the low-level protocol burden

Middleware is a term used to denote products that provide software layers to shield the complexity of application development from core underlying protocols or services. Sample middleware’s includes session oriented, RPC based, message oriented, database interface systems and file transfer products from a variety of vendors.

Communication Techniques

There are generally four mechanisms for managing client to server interaction.

Remote procedure call
Native communication

Messaging

Object Orientation

Remote Procedure calls

The quickest means of developing client server application code is with remote procedure calls. RPCs define a plan for taking standalone programs and converting them to include client server components. By distributing some functions to the server machine, RPCs are a natural mechanism for client developers.

Actually the hand coding RPCs is a difficult process made much easier with the use of precompilers. Products such as NobleNet Incs. Easy RPC provide client programmers with an extremely quick method for distributing functions between client and server systems. These products scan source files and manage the distribution of application functions. The impact of client server coding is thus minimized.

The major drawback to RPCs is that traditionally they have been very synchronous in nature. That is when a client makes RPC requests of the server procedure it is blocked until the server completes the function and returns. These limitations have been removed with the advanced RPC products that provide asynchronous callbacks. Such callbacks are used to post the developer when an RPC has been completed.

Native communication protocols
Traditional client server application developers have been forced to manually support point to point communications between client and server nodes. This usually involves crafting a network interface component that specifically deals with session oriented communications protocols. Application components are required to advertise services, create communications links, send and receive data, and tear down logical connections.

All of these requirements of the application developer are made worse when multiple network protocols are to be supported. Many early client server developers had to provide specific protocol module support for each transport to be used. A great deal of programming and maintenance effort is required to support such interfaces.

Today middleware products provide a single, simplified interface for developers to support.

This simple interface is ported to support many communications protocols and operating systems.

**Messaging**

An extremely popular mechanism for client server interaction is the use of message-oriented products. These products send communications between the client and server in the form of messages. The messages are usually processed by some queuing mechanism at the receiving node. By definition message oriented products are very asynchronous in nature and behave accordingly. Messages are very flexible in that they may be forwarded to other systems or stored for delayed processing. In addition message oriented products remove the effort of underlying network protocols support from the developer.

Message oriented communications is a natural extension of the event driven, message based environments found in many of today’s popular client operating systems. As such there are many products that deal with the message oriented nature of such systems. With message-oriented products the programming interface can be very simple.
First off, access to a message queue is obtained and once established, messages can flow freely over this message queue through a relatively simple API to queue and dequeue messages. More sophisticated products provide extended capabilities such as queue ordering, priority movement, dispatch and execution. Applications wait on these message queues in a similar manner as their event driven GUI code.

**Object Orientation**

It is an emerging technology being employed for distribution of application.

Object model well to the client server environment. The Object Management Group has provided a specification called Common Object Request Broker Architecture that is specially designed to deal with issues of distributing object technology and interfaces.

Separate distributed object models are available from major corporations such as Distributed System Object Model from IBM and Common Object Model from Microsoft.

**Communication Protocols**

In an UNIX system TCP/IP or TLI would be natural choices for a transport mechanism. On UnixWare or NetWare, IPX/SPX would be a native mechanism to adopt. With Windows NT and OS/2, NetBIOS or Named pipes would be a logical choice.

**Client Server Application Interaction Protocols**

Designing an efficient and flexible protocol for client server application level communication is imperative. Generally there two theories of design for application interaction protocols: predefined and flexible. Client server components usually have intimate knowledge of the other. As such a predefined protocol could be established between components where the order and modification of data passed between wouldn’t change. Or a flexible protocol could be defined where there was not a predetermined format to the data being transmitted. This flexible protocol would have to describe the data as it flows between systems.
**Predefined versus Flexible protocols**

A predefined application interaction protocol forces the client application to adhere. With a flexible protocol client data can be accepted and passed between systems as is. A predefined protocol offers an advantage in speed and simplicity. But it makes increased demands on the client program. Predefined protocol could be established between components where the order and modification of data passed between wouldn’t change. Flexible protocol could be defined where there was not a predetermined format to the data being transmitted. The flexible protocol would have to describe the data as it flows between systems.

A flexible protocol is more adaptable to traditional client programming and will require less demands of the client server programmer. Using a flexible protocol client procedure parameters are packaged without the client application having intimate knowledge of client server transaction.

**Packing and Unpacking Protocols**

Efficiency of client server protocols is derived from preparation for, transmission, acceptance, and delivery of data between systems. Most important is the mechanism used by the developer of both client and server for packing and unpacking the data for wire transmission. With flexible protocols, each parameter must be packed and described individually. With a hardwired protocol the data is usually passed between systems in a format that does not require unpacking.

Normally a client application will send a datastructure representing the data between systems ans a server application will receive a pointer to that structure of information. In such protocol no packing and unpacking of parameters is needed.
Predefined Application Protocols

Predefined application level protocols are most prevalent in applications architected from the beginning for client server. These application protocols are very inflexible and as a result are extremely efficient in nature. The reason for this efficiency is that neither the client nor server application components have to pack and unpack parameters for procedures. Instead parameters are placed at the client side into or maintained in the predefined application buffer. At the server procedures are dispatched and passed to predefined application buffer rather than buffer being unpacked into parameters and passed to the procedures.

Flexible Application Protocol

A flexible application level protocol minimizes the impact on the client side application developer. For programmes not familiar with client server design, the flexible application level protocol will greatly ease the amount of effort necessary to produce client server applications. Because a flexible protocol can more easily describe the data being passed as parameters between functions, less coding effort is required of the developer. Less changes are required to package, send, receive and return data from client server components.

A flexible protocol is designed to allow any arbitrary number of arbitrary type parameters to be positioned in a data buffer for transfer between client and server. Seamlessly taking the parameters and converting them to an on-the-wire representation can even be performed by a precompiler. Much as with RPC based products a preprocessor could be developed to automatically convert procedures and parameters into an inter application protocol. The developer would simply need to describe which functions to convert and the characteristics of the parameters.

PREPARING APPLICATIONS FOR CLIENT SERVER

Writing server aware applications are very different from traditional client server, some architectural changes are needed for client server development:
- Identify and group logically similar procedures
- Convert applications into request/response components
- Make responding (server) component reentrant
- Initiate data access on a per client basis

**Identifying and grouping procedures:**

This process should include knowledgeable thinking about how the applications work and what would be most efficient when implemented centrally. Those functions are then target for migration. In addition, it is very important in the identification process to group procedures with like characteristics.

Not modifying newly migrated server components to become aware of their surroundings will likely have a determinately effect. Overhead in the protocols and execution might suffer as a result. It is very inefficient for a client server application to make many independent client to server requests. Grouping procedures as whole components and migrating the entire set to the server will increase efficiency. It also helps to identify the dependencies for the procedural components.

**Converting applications into request/response components:**

The requesting and responding components of client server computing represent the most important architectural change between traditional application programs and client server applications. Each component described in the previous section must be separated through a well defined mechanism. Request/response provides such an environment where the client component makes request of the server components and expects the result to be returned. The component
interaction is thus not being predicted on local procedural interaction, because many times procedural components may not be co located.

When remote procedure call approach is used for client server communication, the procedural components are distributed using a synchronous request/response model. Some other used a message oriented interface for client server interaction. No matter which mechanism is used as the interface to the request/response components, their fundamental value is still the same, to model an interaction between discrete application logic components and mask the physical location of the execution.

**Making functions re entrant:**

Server components may often be executed concurrently on behalf of many clients. In environments where threads are supported, it is very important for the serving components to be re entrant. Being re entrant simply means that an application procedure can be called more than once and executed concurrently without causing corruption. This occurs because a separate stack is used for each calling instance. For instance, many server applications that support threads will dispatch one thread per client connection. Each thread will be executing many similar functions on behalf of the requesting components. Each procedure executed concurrently needs to have re entrant characteristics. Variables defined locally are valid because they exist on a stack allocated uniquely for each call to the procedure. Global variables should be avoided because their modification will change the value contained by other threads making the same procedure call. Making global variables unique per client will alleviate this potential for error.

**Per client data access:**

Per client access will probably provide the greatest code change impact to existing applications as they migrate to client server. When breaking components, it is clear that the responding components will need the code modifications. Each access to global data on behalf of the responding component will need to be made on a per request component basis.
Because of the requirement for server function to be re-entrant, data must either be contained locally or be given serialized access to global data. If global data is used, serialization must be used to coordinate access so as not to corrupt the resource. Per client data is very important for server application components because client data can be referenced outside the scope of a single procedure without inadvertent modification, which is what would happen with globals. Basically a pointer to per client data must be passed on entrance of the re-entrant function. All subsequent access is valid and refers to individual client data, rather than locals or globals.

OPTIMIZING APPLICATIONS FOR CLIENT SERVER

Many of the optimizations available to the developer come as a result of being “client server aware”, minimizing the amount of data sent between client and server is of the utmost importance. This can be achieved in 2 ways. First, by reducing the number of parameters or amount of data. This is done by sending only what is necessary to the server. Second, the server could maintain similar data passed by the client on each function call. State information that travels between client and server is a good candidate to be maintained on the server.

Minimizing the data passed between client and server:

Minimizing this transmission will ensure minimal impact of the application user and restricted use of the physical network, both of which are extremely important. First and foremost, transferring large amounts of data over a network increases the load of the physical infrastructure. The bandwidth of a physical network is a static resource and should not be consumed with wasted or unneeded data. While networks today are rapidly increasing in bandwidth and capacity, it is still not advisable to overuse this precious resource. Applications are slowed by the processing of the large amounts of data being transferred. All parameters in a flexible protocol need to be packed and copied into a buffer at the client, and unpacked and
copied into parameters at the server. This is an expensive process made worse when handling unnecessary data.

**Minimizing data at the server:**

Another way to minimize the amount of data being passed between client and server is by maintaining similar data at server. For client server connections that involves many interactions between client and server, certain code information may be transmitted on each call. Generally, this state information could be easily removed from the wire if the server remembered the data between calls from the client. By maintaining this data at the server, the efficiency of the client server application is increased.

**Processing predefined application protocols:**

Implementing and swift processing of predefined application protocols is sure to improve performance of server side applications. They are geared for high performance by a two fold mechanism. First, because they are inflexible, a minimal amount of time is spent messaging parameters into communications buffers. Second, they may be efficiently processed by the server application if properly architected. In fact, even a flexible application level protocol can be made much more efficient at the server with good coding techniques.

**EXAMPLE CLIENT SERVER IMPLEMENTATIONS**

Once you accept the client server mindset, the process of architecting and developing applications becomes rather easy. There is a wall of acceptance between the requesting and responding components. This new architecture for applications can be easily achieved by creating application functions and procedures as self contained components. Each component can only interact with others through this well defined interface. Self contained application components are a requirement for client server programming. For a client server interaction, there are five main characteristics:

- The interface between request/response components

---

*Dept. Computer Science, SJCET, Palai*
- The formulation of requests form the client side
- The transmission of requests between components
- Acceptance and unpacking of requests form the client
- The execution of requests at the server side

First an interface needs to be defined between requesting and responding components. This interface will vary widely based on the communications technique being implemented. Requests then need to be broken down at the client, communicated to server, accepted, and executed. The formulation and acceptance of requests will vary based on implementation. The core transmission and execution requirements will remain relatively similar no matter what interface is chosen between client and server components.

It is also important to note that for client server communications to take place in as secure environment, some form of authentication of client must be performed. This authentication will be performed when initial contact is made between the client and server. This contact (or location of services) will be initiated by the client application programmer and authenticated by the server.

**Client server interface using procedures:**

A procedure makes a request of another to provide some service of which it expects results to be returned. There is a clear definition of the request and response components, and the only overlap of the two is the parameters they pass and return. These responding procedures can then be migrated to server for processing.

**Request/response interfaces:**

Listing 2.1:

```c
#include...

Main()
{
```
........
CallMe1();
CallMe2();
}

Listing 2.2:
#include….
#define CALL_ME_1 1
#define CALL_ME_2 2
Main()
{
........
CSInterface(CALL_ME_1);
CSInterface(CALL_ME_2);
}

Listing 2.2 & 2.2: traditional application program and interface modification for client server:

Listing 2.2: 

Listing 2.1 is a traditional application program making local procedure calls to CallMe1() and CallMe2(). Listing 2.2 shows the same example using a single function as the interface for the client, thus masking the execution of these procedures on the server. Instead of making direct local function calls, requests will be routed through the CSInterface() procedure interface. Note that the first parameter passed to the CSInterface() procedure is the programmer specified language to be used between client and server. Usually these functions are passed as symbolic constants to signify which function is to be executed at the server. When arriving at the server, these constants will again be used to determine which procedure to execute.

Listing 2.3: client server interface with parameters
#include….
#define CALL_ME_1 1
#define CALL_ME_2 2
Main()
```c
{int num;
long dataVal;
………..
CSInterface(CALL_ME_1,num);
CSInterface(CALL_ME_2,dataVal,num);
}

**Client server protocol:**
Flexible application level protocols must be used for a procedural request/response interface. These protocols allow versatility by describing the data contained in the data packet between client and server.

**Communications of request/response:**
A client must be able to send and receive form the server and the server must be able to receive form the client and return results. The communications interface must provide these generic services to the application layers while remaining transport independent at the bottom layer.

**REQUEST ACCEPTANCE AND DISPATCHING**
Request acceptance and dispatching is an extremely important job of the server interface component. Much of the processing overhead of client server computing exists in this phase. Requests will be accepted from the client and funneled to the server dispatching procedure. With this scenario, each request will be handled with and equivalent CSInterface() at the server side called CS_ServerInterface(). The server component of the client server interaction must have an efficient mechanism for accepting and dispatching server requests. This code will be executed many times from many different requesting stations.

Server interface is performed much the same way CSInterface() provides the client side interface. A case statement is generally used to dispatch requests based on an incoming hdr.requestType command. Parameters will be unpacked and passed directly to the server procedure required to perform work on behalf of the client node.
Listing 1.1

CSServerInterface()
{
  REQHEADER *hdr;
  PARMTYPE *parm;
  Do {
    ReceiveClientRequest(hdr);
    Switch (hdr.requestType) {
      Case CALL_ME_1:
        Arm=hdr+sizeof(REQHEADER);
        Do {
          //unpack parameter (int num)
          //copy to local parameters based on parm.parmType
          // advance to next parameter
          parm=parm+sizeof(PARMTYPE)+parm.parmLength;
          }while(n<hdr.numParmsSent);
        do {
          // copy form local parameter
          //pack parameter n
        } while(n< request.numParmsReturned);
        RespondToClient(hdr);
        Break;
      Case FUNCTION_2:
        Break;
        }while(!error);
    }
  server_CALL_ME_1(int num)
  {
    //process application logic with parameters passed
  }
Listing 1.1: server dispatching procedure:

The CSERVERInterface() performs similar functions to the CSInterface() at the client node. As requests are received via ReceiveClientRequest(), they are processed based on requestType. The corresponding component of the CSERVERInterface() must know how to unpack the parameters and execute the server procedure. The process of return data transmission to the server may actually take place inside the requesting server procedure.

EXECUTION OF REQUESTS

Execution of procedural requests in a server component is very straightforward. The original function prototype and interface is generally intact, and execution of the request consists of merely processing application logic.

Listing 1.2
Server_CALL_ME_1 (WORD clientIndex, int num)
{
    // access per client data via clientStruct[clientIndex]
    // process application logic with parameters passed
}

Server execution with clientIndex:
The listing 1.2 documents the additional requirements to support access to client specific data. The most important difference is that, what point the parameters are packed and sent for return. After executing the server based request, the results must be returned from the server to the client. The code for this return may actually take place within the distributed server procedure or by the dispatching procedure upon return form the server call. This design choice is usually predicated on whether each client request is being serviced by different threads. In such cases, the server packaging and return of data must be done within the executing server component, as
there is no return available form thread creation in the dispatching component. A dedicated thread might also be used to provide a means of returning reply data to the client.

CLIENT SERVER INTERACTION USING MESSAGES

Message based interaction is perhaps the best choice for client server. The interaction mechanisms remain similar. Event loops and actions based upon them is the native model for many of these client operating environments. The major difference is that the application components and the message communication itself may be distributed over the network to a server.

The Request/Response Interface:

This interface is ideal for a message based, client server interaction by implementing messaging as the vehicle for communication. Here asynchronous callbacks are introduced. This is due to the synchronous nature of the interface. Messages are asynchronous and inherently Request/Response in nature. The listing below shows a message interface.
Listing 1.3:
#include…
#define CALL_ME_1 1
#define CALL_ME_2 2

main()
{
………..
MakeRequest(serverQueue, CALL_ME_1);
MakeRequest(serverQueue, CALL_ME_2);
……….. // process other application logic
msg=GetResponse(serverQueue, CALL_ME_1);

msg=GetResponse(serverQueue, CALL_ME_2);
}

The client application can make requests to the serving component in the form of MakeRequest() messages. MakeRequest() requires a serverQueue, a requestType, and optionally a structure used for application level data. Control is immediately returned from MakeRequest() to the client, and the second MakeRequest() is issued. GetResponse() is later used to process the results of the interaction. This asynchronous execution, while very powerful and flexible, must be used with caution.

It is often necessary for messages to operate synchronously. The code below depicts the changes necessary to allow for such operation.

Listing 1.4:
#include…
#define CALL_ME_1 1
#define CALL_ME_2 2
main()
{
…………
MakeRequest(serverQueue, CALL_ME_1);
// get synchronous response
msg=GetResponse(serverQueue, CALL_ME_1);
MakeRequest(serverQueue, CALL_ME_2);
// get synchronous response
msg=GetResponse(serverQueue, CALL_ME_2);
}

Formulating requests:

The most striking difference in the interface between procedural and message based, is in the parameters passed. Acceptance of parameters with a message based approach consists of a
single data block being passed to the MakeRequest() procedure. Management and manipulation of this structure is application controlled and may require client modifications.

Listing 1.4: enhanced MakeRequest() interface

```c
#include…
#define CALL_ME_1 1
#define CALL_ME_2 2

main()
{
    ............
    MakeRequest(serverQueue, CALL_ME_1, msg);
    MakeRequest(serverQueue, CALL_ME_2, msg);
    ............ // process other application logic
    msg=GetResponse(serverQueue, CALL_ME_1);
    msg=GetResponse(serverQueue, CALL_ME_2);
}
```

The listing above shows client passing data via MakeRequest(). MakeRequest() interface has been expanded to include user defined data passed between client and server components.

**Client server interaction protocol:**

The client server interaction protocol using messages as the request/response mechanism must be predefined. Message interaction requires a structured information packet to be written or read. This models very well to predefined application level protocols that have a similar requirement. As such, coding requires knowledge of structures passed as messages. The requesting and responding components only need to know the format of the messages on request and additionally on return.
Listing 1.5:
Typedef struct userData MSG;
Typedef struct request{
WORD clientId; // C: filled in by comm. layer
WORD requestType; // C: function to execute at server
LONG returnCode; // S: return code from server operation
MSG csData; // CS: message component passed/returned
} REQUEST;
C: filled in by client
S: filled in by server

Listing 1.5 is the structure for predefined protocol. It includes request and response information pertaining to the client server interaction as well as the program defined data area called csData. This area will be filled with the requested message for server transmission.

Communications of request/response:

Communications requirements for message or procedural based interaction area are very similar. Application interaction should be removed from the underlying intrinsics of network programming. Communication modules for client server must provide base support such as send and receive to the CSMsgInterface(). This process may be coded form scratch as is necessary for many systems, or a distributed message oriented middleware product could be used for this transfer. 4 or 5 verb API’s to manage message communications are available instead of coding all those by hand. a named queue interface to both client and server is used here. All subsequent communication between the components is based on directly en-queuing and de-queuing the distributed mechanism.

MODULE 3

COMMUNICATION

Multiprogramming
Capability of an OS to support multiple applications running concurrently. It means that an OS can run many applications at the same time. Only one application can actually use a processor at a time, but the multiprogramming OS is responsible for dividing the processor’s execution time and sharing the processor between many applications. In a multiprocessor system OS schedules processes to maximize efficiency of available CPUs.

**Multitasking**

It gives power to users by allowing them to run multiple applications at once. Applications are loaded into memory and appear to the user to be running at the same time. The overall system becomes more efficient because the applications are running concurrently and the processor is kept more active.

**I/O and CPU bursts**

CPU sits idle during I/O operations; efficiency is increased if some work is done during those I/O operations. With multitasking OS reschedules waiting programs to run during I/O operations. The overall throughput of the system is reduced due to concurrent execution of programs.

**Single Tasking versus Multitasking System**

In a single tasking system 2 applications will run back to back, with multitasking systems application waiting for IO will be placed on an IO queue and the other program will begin using the CPU. This mechanism will continue until both jobs are completed, thus increasing the overall capacity of the system.

A single user single tasking OS is burdened by only having one unit of execution. Eg. DOS, for instance, is a single tasking operating system capable of only running one application at a time.

Eg: Novell Netware, UnixWare, Microsoft Windows NT, and IBM OS/2 are all multitasking. Each OS gives the development. In addition to rubbing multiple applications together, these platforms also provide a means by which each application can have multiple executing tasks,
known as processes or threads. The process is represented by an application loaded into memory. Threads are a finer granularity, more efficient mechanism for executing concurrently.

**Processes**

It is made up of a code execution and resource ownership. The system becomes more responsive and fully utilized when processes are multitasked. Additional processes may be created by the application to provide concurrent execution within a program.

**Child and parent processes**

The created processes appear logically as child processes and may inherit access to variables, handles, and resources of the creating process. The creating process is known as the parent process and usually maintains common data and resources. This data may be private to the process, inherited by the child process, or shared as a system wide resource. Various mechanisms are provided by each operating system for inheriting data and resources in child processes.

**Advantages and drawbacks of multiple processes**

Applications architected to take advantage of multiple processes will certainly see performance gains. Features and functions may be distributed among multiple cooperating processes, thereby enhancing execution. A database server might construct one process to communicate with clients, one to read databases, and one to write databases. These processes run concurrently in the system and share the use of the processor. The system is thus efficient.

While processes may increase efficiency their use should be tempered by the amount of system resources available. Processes are useful for segments of applications that may be operated concurrently, but because processes own data and resources, they may become large and inefficient if overused.

**THREADS**

A thread, sometimes called a **lightweight process (LWP)**, is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set and a stack. It shares with other threads belonging to the same process its code section, data section and other operating system resources, such as open files and signals. A traditional (or **heavy weight**) process has a single
thread of control. If the process has multiple threads of control, it can do more than one task at a time.

Much software like Word processor, Web browser etc. are examples of multithreaded programs.

**Benefits**

1. Responsiveness
2. Resource sharing
3. Economy
4. Utilization of multiprocessor architectures

**User and Kernel threads**

Support for threads may be provided at either the user level, for *user threads*, or by the kernel, for *kernel threads*. 
➢ **User threads** are supported above the kernel and are implemented by a thread library at user level. The library provides support for thread creation, scheduling, and management with no support from the kernel. Because the kernel is unaware of user-level threads, all thread creation scheduling are done in user space without the need for kernel intervention. Therefore, user-level threads are generally fast to create and manage, they have drawbacks, however. For instance, if the kernel is single-threaded, then any user-level performing a blocking system call will cause the entire process to block, even if other threads are available to run within the application.

➢ **Kernel threads** are supported directly by the operating system. The kernel performs thread creation, scheduling, and management in kernel space. Because thread management is done by the operating system, kernel threads are generally slower to create and manage than are user threads. However, since the kernel is managing the threads, if a thread performs a blocking system call, the kernel can schedule another thread in the application for execution. Also, in a multiprocessor environment, the kernel can schedule threads on different processors. Most contemporary operating systems—including Windows NT, Windows 2000, Solaris 2, BeOS, and Tru 64, UNIX (formerly Digital UNIX)—support kernel threads.

---

**Multithreading Models**

Multithreading models are:

**Many-to-one Model**

![Diagram of Many-to-one Model](image-url)
Kernel Thread

The Many-to-one Model maps many user level threads to one kernel thread. Thread management is done in user space, so it is efficient, but the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run parallel on multiprocessors.

One-to-one Model

The One-to-One model maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call; it also allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread. Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system. Windows NT, Windows 2000, and OS/2 implement the one-to-one model.

Many-to-Many Model
Kernel threads

This model multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine. Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution. Solaris2, IRIX, HP-UX, and Tru 64, UNIX support this model.

**MODULE 4**

**Scheduling Implementation**

**Scheduling Implementation**

The objective of multiprogramming is to have some process running at all times, in order to maximize CPU utilization. Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues.

**Preemptive Scheduling**

CPU scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state.
2. When a process switches from the running state to the ready state.
3. When a process switches from the waiting state to the ready state.
4. When a process terminates.

When scheduling takes place only under circumstances 1 and 4, the scheduling scheme is non-preemptive; otherwise preemptive.
Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.

First-Come, First-Served Scheduling

It is the simplest. In this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.

The average waiting time under FCFS is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given in milliseconds:

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>24</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
</tr>
</tbody>
</table>

If the process arrive in the order P1, P2, P3 and are served in FCFS order, we get the result as shown in the following Gantt chart:

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td>27</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The waiting time is 0 milliseconds for process P1, 24 milliseconds for process P2, and 27 milliseconds for process P3. Thus the average waiting time is \((0+24+27)/3=17\) milliseconds. If the processes arrive in the order P2, P3,P1, the result will be

<table>
<thead>
<tr>
<th></th>
<th>P2</th>
<th>P3</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>30</td>
</tr>
</tbody>
</table>
The average waiting time now is \((6+0+3)/3=3\) milliseconds. This reduction is substantial. Thus average waiting time under a FCFS policy is generally not minimal and may vary substantially if the process CPU-burst times vary greatly.

**Shortest-Job-First Scheduling**

This algorithm associates with each process the length of the latter’s next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie. The scheduling is done by examining the length of the next CPU burst of a process, rather than its total length.

Consider the set of processes, with the length of CPU-burst time given in milliseconds:

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst-Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>6</td>
</tr>
<tr>
<td>P2</td>
<td>8</td>
</tr>
<tr>
<td>P3</td>
<td>7</td>
</tr>
<tr>
<td>P4</td>
<td>3</td>
</tr>
</tbody>
</table>

Gantt chart

<table>
<thead>
<tr>
<th></th>
<th>P4</th>
<th>P1</th>
<th>P3</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The waiting time is 3 milliseconds for process P1, 16 ms for process P2, 9 ms for process P3 and 0 ms for process P4. Thus the average waiting time is\((3+16+9+0)/4=7\) ms .If we were using the FCFS, then the average waiting time would be 10.25 ms.

The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. By moving a short process before a long one, the waiting time of the short process decreases more than it increases the waiting time of the long process. Thus the average waiting time decreases.
The real difficulty with the SJF algorithm is knowing the length of the next CPU request. For long-term scheduling in a batch system we can use as the process time limit that a user specifies when he submits the job. Thus users are motivated to estimate the process time limit accurately since a lower value may mean faster response. SJF is used frequently in long term scheduling.

The SJF algorithm may be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue while a previous process is executing. The new process may have a shorter next CPU burst than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling.

AS an eg consider the following fur process with the length of CPU burst time given in ms:

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>P4</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

If the processes arrive at the ready queue at times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the Gantt chart:

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P4</th>
<th>P1</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>17</td>
</tr>
</tbody>
</table>

Process P1 is started at time 0, since it is only process in the queue. Process P2 arrives at time 1. The remaining time for process P1(7 ms) is larger than the time required by process P2(4 ms), so process P1 is preempted, and process P2 is scheduled. The average waiting time for this eg is ((10-1)+(1+1)+(17-2)+(5-3))/4=6.5 ms.
A non-preemptive SJF scheduling would result in an average waiting time of 7.75 ms.

Priority Scheduling
A priority is associated with each process, and the CPU is allocated to the process with highest priority. Equal priority process is scheduled in FCFS order.

An SJF algorithm is simply a priority algorithm where the priority \((p)\) is the inverse of the next CPU burst. The larger the CPU burst, the lower the priority and vice versa.

As an example consider the following set of process assumed to have arrived at time 0, in the order P1, P2, …, P5, with the length of the CPU burst time given in ms:

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>P5</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Using priority scheduling we would schedule these processes according to the following Gantt chart:

The average waiting time is 8.2 ms.

Priorities can be defined either externally or internally. Internally defined priorities use some measurable quantity to compute the priorities of a process. External priorities are set by criteria that are external to the OS, such as the importance of the process, the type and amount of funds being paid for compute use.

Priority scheduling may be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the currently running process. A non-preemptive priority-scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithm is indefinite blocking (starvation). A process that is ready to run but lacking the CPU can be considered blocked-waiting for the CPU. A priority scheduling alg can leave some low priority process waiting indefinitely for the CPU.
A solution to the problem of indefinite blockage of low priority process is aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for time. For example, if priorities is ranges from 127(low) to (high), we would decrement the priority of a waiting process by 1 every 15 minutes. Eventually even process with an initial priority in the system and would be expected.

Round-Robin Scheduling

The round-robin (RR) scheduling algorithm is designed especially for time-sharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a time quantum (or time slice) is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as circular queue.

To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

The average waiting time under the RR policy however is often quite long. Consider the following set of processes that arrive at time 0, with the length of CPU-burst time given in ms:

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>24</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
</tr>
</tbody>
</table>

If we use a time quantum of 4 ms, then process P1 gets the first 4 ms, then process P1 gets the first 4 ms. Since it requires another 20 ms, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2. Since process P2 does not need 4 ms, it quits before its time quantum expires. The CPU is then given to the next process, process
P3. Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum. The resulting RR schedule is:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>P2</td>
<td>P3</td>
<td>P1</td>
<td>P1</td>
<td>P1</td>
<td>P1</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>14</td>
<td>18</td>
<td>22</td>
</tr>
</tbody>
</table>

The average waiting time is \( \frac{17}{3} = 5.66 \text{ ms} \).

In the RR scheduling alg, no process is allocated the CPU for more than 1 time quantum in a row. If a process CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The R scheduling alg is preemptive.

If there are \( n \) processes in the ready queue is \( q \), then each process gets \( \frac{1}{n} \) of the CPU time chunks of at most \( q \) time units. Each process must wait no longer than \( (n-1)*q \) time units its next time quantum.

The performance of the RR scheduling of the RR scheduling depends heavily on the size of the time quantum. At one extreme, if the time quantum is very small the RR approach is called processor sharing, and appears to the users as though each of \( n \) processes has its own processor running at \( 1/n \) the speed of the real processor.

Multilevel Queue Scheduling

A common division is made between foreground (or interactive) processes and background (or batch) processes. A multilevel queue scheduling alg partitions the ready queue into several separate queues (fig). The process are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority or process type. Each queue has its own scheduling alg.

Highest priority

System process

Interactive process

Interactive editing process
Batch process

Student process

Lowest priority

Fig Multilevel queue scheduling.

Each queue has absolute priority over lower priority queues. No process in the batch queue, for ex. could run unless the queue for system processes, interactive process, and interactive editing process were all empty. If an interactive editing process entered the ready queue while a batch process was running the batch process would be preempted.

Multilevel Feedback Queue Scheduling

In this the process are permanently assigned to queue on entry to the system. Process does not move between queues. If there are separate queues for foreground and background processes, for ex processes do not move from one queue to the other, since process do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but the disadvantage of being inflexible.

This allows a process to move between queues. If a process uses too much CPU time it will be moved to a lower priority queue. This scheme leaves I/O bound and interactive processes in the higher priority queue.

For ex consider a multilevel feedback queue scheduler with three queues numbered from 0 to 2 (fig). The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly processes in queue 2 will be executed only if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue. A process that arrives for queue 0 will in turn preempt a process in queue 1.

The Critical-Section Problem

Consider a system consisting of n processes \{P0,P1,...,Pn-1\}.Each process has a segment of code called critical section, in which the process may be changing common variables,
updating a table, writing a file and so on. The important feature of the system is that, when one process is executing in its critical section. Thus the execution of critical sections by the process is mutually exclusive in time. The critical section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section.

do
{
  entry section

  critical section

  exit section

  remainder section
} while(1);

fig 1.1. General structure of typical process Pi.

A solution to the critical-section problem must satisfy the following three requirements:

1. Mutual Exclusion: If process Pi is executing in its critical section, then no other process can be executing in their critical sections.

2. Progress: If no process is executing in its critical section and some processes wish to enter their critical sections, then only those process that are not executing in their remainder section can participate in the decision on which will enter its critical section next and this selection cannot be postponed indefinitely.

3. Bounded waiting: There exists a bound on the number of times that other processes are allowed to enter its critical section and before that request is granted.
Two-Process Solutions

In this section, we restrict our attention to algorithms that are applicable to only two processes at a time. The processes are numbered P0 and P1. For convenience, when presenting Pi, we use Pj to denote the other process; i.e., j = i - 1.

Algorithm 1:

Our first approach is to let the processes share a common integer variable 'turn' initialized to 0 (or 1). If turns == 1, then process Pi is allowed to execute in its critical section. The structure of process Pi is shown below.

\[
\text{do}
\begin{align*}
\text{while (turn! = i)} & \\
\text{critical section} & \\
\text{turn} = j; & \\
\text{remainder section} & \\
\text{while (1)} & \\
\end{align*}
\]

fig 1.2. The structure of process Pi in algorithm 1

This solution ensures that only one process at a time can be in its critical section. However, it does not satisfy the progress requirement, since it requires strict alternation of processes in the execution of the critical section. For ex, if turns == 0 and P1 is ready to enter its critical section, P1 cannot do so, even though P0 may be in its remainder section.

Algorithm 2

The problem with algorithm 1 is that it does not retain sufficient information about the state of each process; it remembers only which process is allowed to enter its critical section. To remedy this problem, we can replace the variable “turn” with the following array:

Boolean flag[2];
The elements of the array are initialized to false. If flag [I] is true, this value indicates that Pi is ready to enter the critical section. The structure of process Pi is shown in fig3.

In this alg, process Pi first sets flag [I] to be true, signaling that it is ready to enter its critical section. Then Pi checks to verify that process Pj is no also ready to enter its critical section. If Pj were ready, then Pi would wait until Pj had indicated that it no longer needed to be in the critical section (ie, until flag [j] was false). At this point Pi would enter the critical section. On exiting the critical section, Pi would set flag [I] to be false, allowing the other process (if it is waiting) to enter its critical section.

```plaintext
do
{
    flag[I]=true;
    while (flag[j]);
}

Critical section
Flag[I]=false;

Remainder section
}while(1);
```

Fig3. The structure of process Pi in alg2

Semaphores

The solution to critical section problem are not easy to generalize to more complex problems. To overcome this difficulty, we can use a synchronization tool called a semaphore. A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait and signal. The classical definition of wait in pseudo code is

wait(S)
{while(S<=0)
    ; // no-op
    S--;
}

The classical definition of signal in pseudo code is

Signal(S){
    S++;
}

Modifications to the integer value of the semaphore in the wait and signal operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of the wait(S), testing of the integer value of S(S<=0), and its possible modification (S--), must also be executed without interruption.

Usage

We can use semaphore to deal with the n-process critical section problem. The n processes share a semaphore, mutex(standing for mutual exclusion), initialized to 1. Each process Pi is organized as shown in fig4.

We can also use semaphores to solve various synchronization problems. For eg, consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2. Suppose that we require that S2 be executed only after S1 has completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0, and by inserting the statements in process P1 and the statements

Wait(synch);
S2;
in process P2. Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal(synch), which is after S1.
S1;
Signal(synch);
Do
{
    Wait(mutex);
    Critical section
    Signal(mutex);
    Remainder section
}while(1);

Fig 4. Mutual exclusion implementation with semaphores

Semaphore Implementation

The main disadvantage of the mutual exclusion solutions is that they all require busy waiting. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a spinlock (because the process “spins” while waiting for the lock).

Spinlocks are useful in multiprocess systems. The advantage of a spinlock is that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus when locks are expected to be held for short times, spinlocks are useful.

To overcome the need for busy waiting, we can modify the definition of the wait and signal semaphore operations. When a process executes the wait operation and finds that the semaphore value is not positive, it must wait. However, rather than busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process into waiting queue associated with the semaphore, and
the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal operation. The process is restarted by wakeup operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal operation. When such a state is reached, these processes are said to be deadlocked.

To illustrate this, we consider a system consisting of two processes P0 and P1 each accessing two semaphores S and Q set to the value 1:

```
P0          P1
Wait(S);    Wait(Q);
Wait(Q);    Wait(S);
..          ..
..          ..
signal(S);  signal(Q);
signal(Q);  signal(S);
```

Suppose that P0 executes wait(S), and then P1 executes wait(Q). When P0 executes wait(Q), it must wait until P1 executes signal(Q). Similarly when P1 executes wait(S), it must wait until P0 executes signal(S). Since these signal operations cannot be executed, P0 and P1 are deadlocked.

We say that a set of process is in deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release.
Another problem related to deadlocks is indefinite blocking or starvation, a situation where processes wait indefinitely within the semaphore. Indefinite blocking may occur if we add and remove process from the list associated with a semaphore in LIFO order.

Binary Semaphores

The semaphore construct described in the previous sections is commonly known as counting semaphore, since its integer value can range over an unrestricted domain. A binary semaphore is a semaphore with an integer value that can range only between 0 and 1. A binary semaphore can be simpler to implement than a counting semaphore depending on the underlying hardware architecture.

Let S be a counting semaphore. To implement it in terms of binary semaphore we need the following data structures:

- Binary-semaphore S1, S2;
- Int C;

Initially S1=1, S2=0 and the value of integer C is set to the initial value of the counting semaphore S.

The wait operation on the counting semaphore S can be implemented as follows:

```
Wait(S1);
C--;
If(C<0)
{signal(S1);
wait(S2);
}signal(S1);
```

The signal operation on the counting semaphore S can be implemented as follows:

```
Wait(S1);
C++;
If(C<=0)
signal(S2);
else
signal(S1);
```
MODULE 5
NETWORK COMMUNICATION
NETWORK COMMUNICATION

The language of communication, no matter the physical layout, each communicating partner must understand how the other party is communicated, or what “language” is being used. If two partners establish a physical communications link but have no way of knowing what data or format the other partner is sending, the communication is useless. Protocols are designed to solve this problem by establishing a language as a set of rules that both parties understand.

There are many protocols today at all levels of communication, including physical hardware-, network-, and application-level protocols. This section deals mainly with network protocols. It also discusses their functional characteristics, including the underlying hardware, naming, and programming interface.

FUNCTIONAL CHARACTERISTICS OF NETWORK COMMUNICATION

Network communication is meant to facilitate data transmission between any number of stations or nodes. Certain functional characteristics are common to all means of network communication:

- Protocols
- Addressing
- Programming interface

Many protocols are available throughout a network. Attempts have been made to describe these protocols in terms of layers, with the most popular reference model of this type being the Open Systems Interconnection (OSI) model. The application developer must be able to accept names in many forms and convert them to network protocol addresses. In addition, the programmatic interfaces of each individual network protocol must be broad enough to support many features and functions. And programmatic interfaces must be flexible in terms of operating characteristics.

THE OSI REFERENCE MODEL
The International Standards Organization (ISO) has developed a standard layered model for communication called the Open Systems Interconnection (OSI).

With the OSI model, a clear distinction is drawn between one level of protocol and another. Each protocol level has specific functions and characteristics. In theory, their individual duties do not overlap. As a result, categories of protocols have been created, each servicing respective layers in the OSI model.

**Application layer:** Responsibilities include application-to-application interaction protocols. Application layer protocols are usually created by the developer for communicating b/w partner applns, each understanding the language or protocol.

**Presentation Layer:** Allow communications between partners in a syntactically cohesive manner. It is the responsibility of this level to convert underlying data, structures and syntactical differences between nodes communicating on the network.

**Session Layer:** Session layer protocols provide the semantics of a conversation between partner nodes. These protocols decide the organization of the communications.

**Transport Layer:** This level of the OSI model is responsible for providing reliable data communications between nodes on a network. It is quite important because it provides the basis of communications for all upper-layer services.

**Network Layer:** Responsible for managing the operation of the network, specifically for the routing of packets sent between Transport-level products. This layer provides control information for the management of data packets for which it routes.

**Data-link Layer:** Responsible for controlling the exchange of data between Physical layer and the upper-level Network layer. This layer places packet data into frames for the appropriate physical network.
**Physical Layer:** Responsible for the physical transmission of the information over a network. This layer represents anything pertaining to the physical network such as encoding, transmission and topology.

**NETWORK PROTOCOLS**

Network protocols are implemented to allow for the knowledgeable transfer of data between nodes on a network. A protocol is a set of rules that must be understood and followed when communicating between stations. Protocols exist at all levels of the OSI model, from Physical to Application. Shielding one level of protocol from another in the OSI model allows many upper layers to be totally independent of low protocol layers. Many corporations have TCP/IP networks, IPX/SPX networks, AppleTalk networks, and others all installed at one or multiple network protocols may travel over the physical wires.

**THE STRUCTURE OF A NETWORK PROTOCOL**

Network protocols are absolutely necessary for the transfer of information between network nodes. Because these protocols govern the way information is communicated, they must have clear-cut rules of language.

**SUPPORTING UNDERLYING LAYERS**

The layering of protocols is especially important for transparency of service. Each protocol can manage its own features and functions independent of any other layer or service. In addition, modifications or updates can be made without disrupting other layers. Because of this service or protocol independence, supporting the underlying layer is a very simple task.

**Maintaining independence through protocol layering:**

In any event, the layering of protocols provides clear boundaries of duties. While each upper-layer protocol must make a specific implementation to a lower-level protocol, their duties don’t intersect. This allows the protocols themselves to be independent of each other and form the basis for supporting multiple protocols concurrently.
**CONNECTION-ORIENTED Vs. CONNECTIONLESS COMMUNICATION**

Two forms of service are generally associated with network communication:
Connectionless
Connection-oriented

**Connectionless protocols:**
A connectionless protocol is one that provides a best-effort service. Messages are sent, fully self-contained, between partners in a peer environment. These protocols are generally used for high performance, since error checking and flow control are not implemented. As a result, connectionless data grams are not guaranteed, and when arriving at the target partner, are not necessarily in the order in which they were sent.

**Connection-oriented protocols:**
Connection-oriented protocols provide a reliable, guaranteed delivery. End-to-end communications are establishing and a reliable link is guaranteed. This type of service provides for the sequencing of packets and performs error checking and flow control. Some protocols actually avoid the overhead of address resolution for every transfer because there is already an established partner.

**Inter process Communication**

Inter process communication (IPC) is generally used as a communications vehicle for two processes running on the same physical machine. These processes, whether parent-child by relationship or unrelated, may need to communicate with each other by sharing handles, signals, or general application-level data. IPC is very important to consider when writing server applications. Because server applications tend to start many processes, each handling a segment of the application, some form of IPC must be used to communicate between the partner tasks.
Following are some sample inter process communications mechanisms:

**MECHANISM** | **DESCRIPTION**
--- | ---
Pipes | Pipes are used as a simple file-based mechanism for communicating data back and forth between tasks. Pipes may be one-way, where there is one writer and one reader, or bi-directional, where there may be reads and writes by either process.
Messages | Messages provide basic send and receive primitives to transfer data between processes.
Semaphores | Semaphores are used to coordinate access between processes.
Shared memory | Shared memory is the process of defining a section of memory visible to more than one process for storage and retrieval of data.

While IPCs are generally thought of as local to one machine, some mechanisms will work over a network. The most popular protocol for such communications is named pipes. Named pipes are an extension of the pipe interface, except that partners in communication can be located over a physical network. In addition, some middleware vendors such as Momentum Software with xIPC have extended the native IPC mechanisms (messages, semaphores, and shared memory) of many platforms to be distributed over a network as well. In this section, both pipes (anonymous and named) and messages will be examined in depth.

**PIPES FOR COMMUNICATION BETWEEN TWO PARTNER PROCESSES**

Pipes are used for communication between two partner processes. These processes may be physically located on the same machine and communicate via anonymous pipes, or on separate machines and communicate through a named scheme. Anonymous (unnamed) pipes are generally associated with local, non-network access, while named pipes can be operated over a network. Generally, an anonymous pipe is used to communicate between related processes,
while named pipes are used to communicate between unrelated processes (may even on different machines).

Each of the evaluated operating system platforms provide a named pipe interface, except for NetWare. UnixWare provides both anonymous and named pipes and has a unique interface. The named pipe interface for Windows NT and OS/2, however, is almost the same. There are small semantic name differences, such as CreateNamedPipe() in Windows NT and DOS CreateNPipe() in OS/2. In addition, some operational control parameters differ, but the underlying functions are extremely similar. As a result, we will only cover one set of APIs, Windows NT, for pipe interfaces.

**UnixWare Support for Pipes**

UnixWare provides support for both named and anonymous pipes. The standard mechanism for inter process communication is with the use of anonymous pipes. Pipes provide a bi-directional means of communication for these processes. When a pipe is created, UnixWare actually creates two STREAMS and connects them to form the two-way capability. The interface for pipe creation and access are very straightforward because standard file system function semantics are used.

UnixWare returns file descriptors for reading and writing (int fd[2]). Writing to fd[0] becomes data-readable with fd[1], and the same holds true for the reverse. Any process with proper access to these descriptors can use them to access the pipe. Operations on the STREAMS-based pipes are generally file system commands. Reading occurs with the read() or getms g() commands, writing with write() or putms g(), and closing with close(). In addition, other file-system-related calls are available, such as ioctl().

For a stream to become named, it must be attached to a node in the file system. This is done with the fattach() command. Remote clients may access this named stream if they have mounted access to the server’s file system. In such an event, the client need only specify the name of the named pipe to gain subsequent access. Connection to the server pipe will be multiplexed when many remote clients are communicating. It is possible to create a unique pipe for each client requesting access with the use of the connld module on the STREAM-based pipe. In such an environment, subsequent opens from a remote client would cause a new file descriptor to be
created, the server would be notified, and the client would receiver the handle back from the open call.

**UnixWare pipe APIs** Following are descriptions of UnixWare pipe APIs:

**Pipes (file Descriptors [2])**
This function creates an anonymous pipe and returns two file descriptors. Each file descriptor has read and writes access and may be used by any process with appropriate access privileges. For streams to become named, they must be associated with a node in the file system. This is done with the fattach() command.

**Fattach(fileDescriptor, path)**
The fattach command attaches an open file Descriptor with a named node in the local file system. This command will be executed by a server process to give a named access to a pipe. Subsequent open() operations from a client with a mounted directory of the server using fattach() will connect both partners to the same pipe.

**Fdetach(path)** The fdetach() command is used to detach a name from a file system node.

**Windows NT Supprot for Pipes**

Windows NT provides two operating modes for inter process communications via pipes: named and unnamed. Anonymous, or unnamed, pipes are unidirectional in nature. Creating an anonymous pipe returns two handles, one for reading and one for writing. The handles are usually passed to child processes via inheritance. Named pipes, on the other hand, may be either a unidirectional or bi-directional pipe between two processes.

Named pipes are used for inter process communication via a client and a server process (either on one physical machine or over a network). The naming scheme used is the Universal Naming Convention(UNC), where the path name begins with \servername\servicename, in the case of named pipes, the format is \servername\pipe\pipename, access is generally made as follows: The server creates a named pipe, and clients access it via Create-File or CallNamedPipe.
Named pipes are created as instances that may allow many clients to access unique pipes with the same name and transfer data back and forth to the server. The server creates an instance for every unique connection it requires from a client. Access modes and privileges may be associated with a pipe controlling the read-write access, the mode of operation (blocking/no blocking), type of data transfer (message or byte mode), as well as many other options.

**Client Server Applications**

A client server application is a piece of software that runs on a client computer and makes request to a remote server. Many such applications are written in higher level visual programming languages where as forms and most business logic reside in the client application. Often such applications are database applications that make database queues to a remote control database server.

In a database application data related number crunching can occur on the remote database server where the processing is close to physical data.

A client server application can be cross platform if it is written in a cross platform language; or it can be platform specific. Here, there is an advantage that the application can potentially provide a user interface that is native in environment to the user Operating System or platform environment. It is running under client/server application either runs locally or a client computer or through something like terminal server, centric, a VNC can work closely with the underlying as to provide a rich powerful robust interface.

Client/server application is a terminal service environment must be able to handle multiple clients connecting on the same computer.

1. **Network client server applications:-**

   i) **Entrez:**
A client server version of Entrez called Network Entrez provides remote Internet access to the Entrez database residing at the NCVI. To use the networked version direct TCP/IP access to the internet is a mandatory requirement.

Network Entrez client programs are currently available to the following computer platform.

- Macintosh
- MS Windows
- Sun Spark Station
- DEC alpha running OST/1
- DEC Ultrix
- DEC VMS

In addition to the client server version, there is also a version of Entrez available. It is BLAST.

**ii) BLAST (Net BLAST):**
Net BLAST is the newest version of the BLAST client software that access the current version of gapped BLAST at NCBI. The Net BLAST client (BLAST) is available via FTP for a number of popular platforms like Linux, Solaris, SGI, Windows and Macintosh. Net BLAST routes a user’s query to the BLAST server at NCBI that can handle the work in the shortest time rather than to a particular machine. Users receive results back faster and their work is not interrupted by server maintenance on a server.

**II Web Applications:-**

These are analogous to common gateways interface (CGI) applications and consist of dynamically generated web pages accessed through a web browser.

We can create a web application quickly and easily with web objects provides many HTML based elements that we can use to build your web applications interface. These elements, frame
from simple user interface, widgets to elements that provide for the conditional or iterative display of the content.

Web components encapsulate more than the layout of a web page. These also encompass all Java files that links the components into a coherent entity.

When you need the fast and full featured user interface, of desktop client/server application, you can partition your application so that a partition of it including all or part of the user interface logic runs in Java directly and the client. Client server communication is handled by the web objects. The web object applications that are partitioned in this way are known as Java client applications.

Java client distributes the object of your web objects application, between the application server and 1 or more clients typically Java application. It is based as a distributed multi-user client/server architecture where processing duties are divided between a client an application server, a database server and a web server.

With a Java client application you can partition business objects containing business logic and data into a client side and server side. This partition can improve performance and at the same time help to secure legacy data and business rules.

The fig. below indicates a Java client application in which the client partition is running as an application installed on the user’s computer. Java client applications just like web applications can communicate with the application server using HTTP request. Java client passes objects between the partition of your application presiding on the user’s computer and the partition of your application that remains on the application server.
IV  Web Services:-

Web services are an innovative implementation of distributed computing. Web objects allows you to expose class methods as web services provide an efficient way for application to communicate with each other. Based on simple object access protocols,(SOAP) messages that wrap XML documents, web services provide a flexible infrastructure that leverages the HTTP’ or TCP/IP.
Web services provide more than an information exchange system. When an application implements some of its functionality using web services it becomes more than the sum of its parts. You can create a web service operation from another provider to give the consumer information tailored to the needs. Web service operation is similar to the method of a Java class. A provider is an entity that publishes a web service while the entities that use a web service are called consumers. Web application as well as Java client application can take advantage of web services.

Fig. A dynamic publishing website using web services