Interprocess communication in C in Operating Systems

- the C language says nothing about IPC
  - the operating system offers various mechanism for IPC
  - there normally exist a set of library headers and library archives which are provided by the operating system and system C library

- common methods of IPC are:
  - sockets (Berkeley and System V Transport Layer Interface)
  - shared memory (normally used in conjuction with semaphores)
  - semaphores

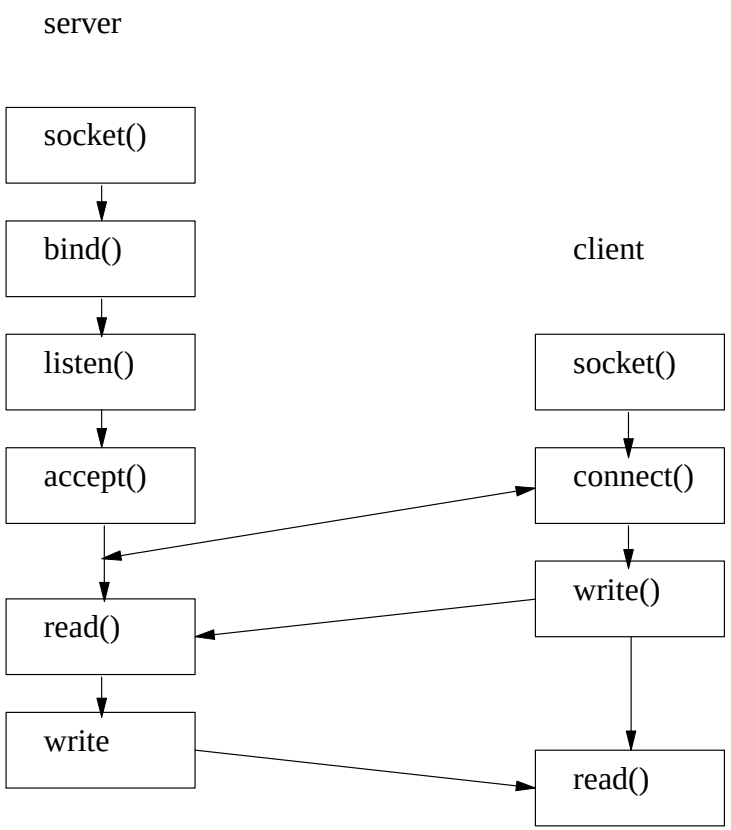
Berkeley Sockets

- the C interface to sockets ultimately gives the programmer a file descriptor on both client and server which can be both read from and written to

- elegant as the user application can map the code onto a socket or a file

Process	Description	Function
server	create end point	`socket()`
	bind address	`bind()`
	specify queue	`listen()`
	wait for connection	`accept()`
client	create end point	`socket()`
	bind address	`bind()`
	connect to server	`connect()`
	transfer data	`read()`
		`write()`
		`recv()`
		`send()`
	datagrams	`recvfrom()`
		`sendto()`
	terminate	`close()`
		`shutdown()`

Connection oriented sockets



Consider Python Code for a Server

```
#!/usr/bin/python

from socket import *
myHost = ""
myPort = 2000

s = socket(AF_INET, SOCK_STREAM)
# bind it to the server port number
s.bind((myHost, myPort))
# allow 5 pending connections
s.listen(5)

while True:
    # wait for next client to connect
    connection, address = s.accept()
    while True:
        data = connection.recv(1024)
        if data:
            connection.send("echo -> " + data)
        else:
            break
    connection.close()
```

Consider Python Code for a client

```
#!/usr/bin/python

import sys
from socket import *
serverHost = "localhost"
serverPort = 2000

# create a TCP socket
s = socket(AF_INET, SOCK_STREAM)

s.connect((serverHost, serverPort))
s.send("Hello world")
data = s.recv(1024)
print data
```

Now the equivalent client C code

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>
#include <malloc.h>
#include <signal.h>
#include <string.h>
#include <stdlib.h>
```

Now the equivalent client C code

```
#if !defined(TRUE)
#  define TRUE  (1==1)
#endif
#if !defined(FALSE)
#  define FALSE (1==0)
#endif

#define ERROR(X)   { printf("%s:%d:%s\n", __FILE__, __LINE__, X); \
                     exit(1); }

#define ASSERT(X)  { if (! (X)) \
                       { printf("%s:%d: assert(%s) failed\n", \
                         __FILE__, __LINE__, #X ); exit(1); }}

#define MAXBUFFER 4096

const char *serverName = "localhost";
const int portNo = 2000;
```

Now the equivalent client C code

```
main ()
{
  struct hostent     *hp;
  struct sockaddr_in  sa;
  int                 socketFd;
  char                inBuffer[MAXBUFFER];

  hp = gethostbyname(serverName);
  if (hp == NULL) {
    fprintf(stderr, "cannot find host: %s\n", serverName);
    exit(1);
  }

  memset((void *)&sa, 0, sizeof(sa));
  sa.sin_family = AF_INET;

  /* memcpy(dest, src, length) */
  memcpy((void *)&sa.sin_addr, (void *)hp->h_addr, hp->h_length);
  sa.sin_port   = htons(portNo);
```

Now the equivalent client C code

```
  /*
   *  Open a TCP socket (an Internet stream socket)
   */

  socketFd = socket(hp->h_addrtype, SOCK_STREAM, 0);
  if (socketFd < 0)
    ERROR("failed to connect to the TCP server");

  if (connect(socketFd, (struct sockaddr *)&sa, sizeof(sa)) < 0)
    ERROR("failed to connect to the TCP server");

  write(socketFd, "hello world\n", 12);
  read(socketFd, inBuffer, sizeof(inBuffer));
  printf("result from server = %s", inBuffer);
  close(socketFd);
}
```

Now the equivalent server C code

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>
#include <malloc.h>
#include <signal.h>
#include <string.h>
#include <stdlib.h>
```

Now the equivalent server C code

```
#if !defined(TRUE)
#  define TRUE  (1==1)
#endif
#if !defined(FALSE)
#  define FALSE (1==0)
#endif

#define ERROR(X)   { printf("%s:%d:%s\n", __FILE__, __LINE__, X); \
                     exit(1); }

#define ASSERT(X)  { if (! (X)) \
                      { printf("%s:%d: assert(%s) failed\n", \
                        __FILE__, __LINE__, #X ); exit(1); }}

#define MAXBUFFER 4096

const int portNo = 2000;
const char *serverName = "localhost";
```

Now the equivalent server C code

```
main ()
{
  struct hostent    *hp;
  struct sockaddr_in sa;
  struct sockaddr_in isa;
  int socketFd;
  int a, b, r, i;
  char  inBuffer[MAXBUFFER];

  hp = gethostbyname(serverName);

  /*
   * Open a TCP socket (an Internet stream socket)
   */

  socketFd = socket(hp->h_addrtype, SOCK_STREAM, 0);
  if (socketFd < 0)
    ERROR("socket");
```

Now the equivalent server C code

```
memset(&sa, 0, sizeof(sa));
ASSERT((hp->h_addrtype == AF_INET));
sa.sin_family      = hp->h_addrtype;
sa.sin_addr.s_addr = htonl(INADDR_ANY);
sa.sin_port        = htons(portNo);

b = bind(socketFd, (struct sockaddr *)&sa, sizeof(sa));
if (b < 0)
  ERROR("bind");

listen(socketFd, 1);
```

Now the equivalent server C code

```
  while (TRUE) {
    i = sizeof(isa);
    a = accept(socketFd, (struct sockaddr *)&isa, &i);
    if (a < 0)
      ERROR("accept");
    r = read(a, inBuffer, MAXBUFFER);
    r = write(a, "echo -> ", 8);
    r = write(a, inBuffer, strlen(inBuffer));
    close(a);
  }
}
```

Coursework clientlib for C

- given that the Python client code was so simple it would be good to create a simple client side library which can be used to connect to a server

- for example

Coursework test code

```
#include <stdio.h>
#include "clientlib.h"

main()
{
  cli *c;
  char buffer[50];

  c = clientlib_socket ();
  c = clientlib_connect (c, "localhost", 2000);
  write(clientlib_getfd(c), "hello world", 12);
  read(clientlib_getfd(c), buffer, 50);
  printf("server sent: %s\n", buffer);
  c = clientlib_close(c);
}
```

clientlib.h

```
#if !defined(clientlibH)
#  define clientlibH
#  if defined(clientlibC)
#     define EXTERN
#  else
#     define EXTERN extern
#  endif

typedef void cli;

/*
 *  clientlib_socket - returns a newly malloced cli.
 *                     We dont actually call socket here.
 */

EXTERN cli *clientlib_socket (void);
```

clientlib.h

```
/*
 *  clientlib_connect - creates the socket and
 *                      connects to server, port.
 *                      It returns the cli passed.
 */

EXTERN cli *clientlib_connect (cli *c,
                               const char *server, int port);


/*
 *  clientlib_getfd - returns the file descriptor associated
 *                    with the socket.
 */

EXTERN int clientlib_getfd (cli *c);
```

clientlib.h

```
/*
 *  clientlib_close - closes the socket and frees up
 *                    any allocated space.
 */

EXTERN cli *clientlib_close (cli *c);
```

clientlib.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>
#include <malloc.h>
#include <signal.h>
#include <string.h>
#include <stdlib.h>

#if !defined(TRUE)
#  define TRUE  (1==1)
#endif
#if !defined(FALSE)
#  define FALSE (1==0)
#endif
```

clientlib.c

```
#define ERROR(X)    { printf("%s:%d:%s\n", __FILE__, __LINE__, X); \
                      exit(1); }

#define ASSERT(X)   { if (! (X))
                         { printf("%s:%d: assert(%s) failed\n", \
                            __FILE__, __LINE__, #X ); exit(1); }}


#define clientlibC
#include "clientlib.h"

typedef struct {
  struct hostent *hp;
  struct sockaddr_in sa;
  int socketFd;
} real_cli;
```

clientlib.c

```
/*
 *  clientlib_socket - returns a newly malloced cli.
 *                     We dont actually call socket here.
 */

cli *clientlib_socket (void)
{
  real_cli *c = (real_cli *)malloc (sizeof(real_cli));

  return (cli *)c;
}
```

clientlib.c

```
/*
 *  clientlib_connect - connects to server, port.
 *                      It returns the cli passed.
 */

cli *clientlib_connect (cli *c, const char *server, int port)
{
  real_cli *rc = (real_cli *)c;

  /* you need to complete this */
}
```

clientlib.c

```
/*
 *  clientlib_getfd - returns the file descriptor associated
 *                    with the socket.
 */

int clientlib_getfd (cli *c)
{
  real_cli *rc = (real_cli *)c;

  /* you need to complete this */
}
```

clientlib.c

```
/*
 *  clientlib_close - closes the socket and frees up
 *                    any allocated space.
 */

cli *clientlib_close (cli *c)
{
  /* you need to complete this */
}
```