Lecture Notes for Lecture 12 of CS 5600 (Computer Systems) for the Fall 2019 session at the Northeastern University Silicon Valley Campus.

Inter-Process Communication

Philip Gust,
Clinical Instructor
Department of Computer Science

These slide highlight content from suggested readings.

Review of Lecture 11

- Lecture 11 presented a model of computing where a program can to perform multiple tasks *concurrently*, and coordinate the actions of these tasks.
- The mechanism we studied that accomplishes this is known as a *thread*, and each task is called a thread of execution. A program that operates in this way is called multi-threaded.
- In this lecture we studied how threads work, and how they are implemented in C.
- We also looked at concurrency control mechanisms to synchronization multiple threads. One was a *semaphores*, which uses a counter to lock a *critical section* of code.
- The other was a *monitor*, which combines a lock and a condition variable to guard a resource, such as a queue.

Inter-Process Communication

- In this lecture we will learn about the ways that processes can communicate with each other and the mechanisms available for *inter-process communication*.
- We will consider two situations: when the processes are running on the same computer, and when the processes are running on different computers.
- Two mechanisms are available when running on the same computer: *shared memory* and *named pipes*.
- When running on different computers connected by a network, processes can communicate using *ports* and *sockets*, and through *remote procedure calls* (*RPCs*).

Inter-Process Communication

Local Inter-Process Communication

- There are three mechanisms for IPC among local processes on the same computer
  - Signals
  - Shared Memory
  - Named Pipes

Inter-Process Communication

Signals

- We have already studied signals, an *asynchronous* notification sent by the operating system to a process about an event that occurred.
  - Signals may be generated by the operating system as a result of a low-level condition that is of interest to a given process (e.g. divide by 0, segment violation).
  - A running process can ask the operating system to send a signal to another process as a way to communicate with or to control the target process.
  - A running process can ask the operating system to send a signal to itself as a way for the process to modify or control its own execution (e.g. setting an timer alarm).

Inter-Process Communication

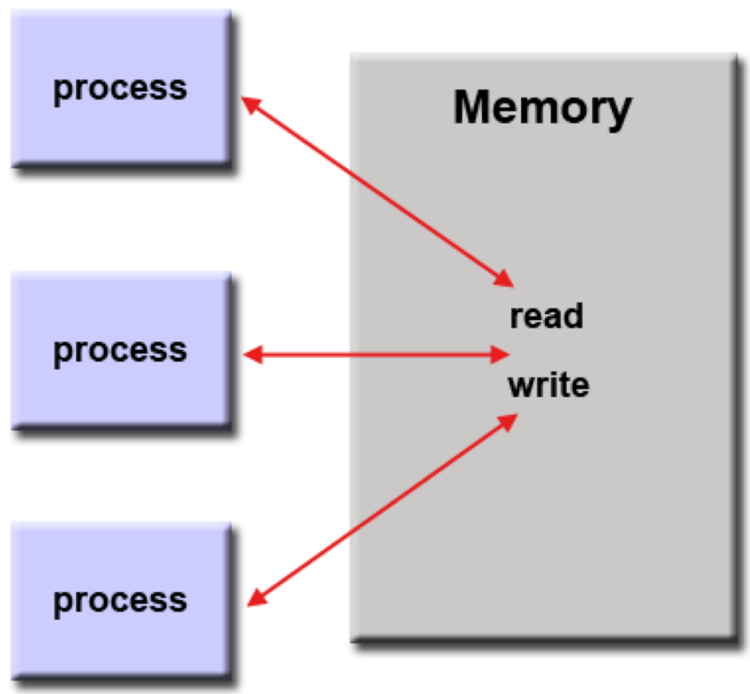
Shared Memory

- Shared memory allows two or more processes to share a given region of memory.
- This is the fastest form of IPC because the data does not need to be copied between communicating processes
- Access to share memory requires synchronizing access to a given region among multiple processes.
  - If the server/producer process is placing data into a shared memory region, the client/consumer process should not try to access it until the server is done
  - Various mechanisms such as locks/semaphores are used to control access to the shared memory, resolve contentions and to prevent race conditions and deadlocks.

Inter-Process Communication

Shared Memory

- This diagram illustrates sharing a segment of memory among several processes



Inter-Process Communication

Shared Memory

- POSIX C library provides APIs managing shared memory.

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>

/**
  * shmget: used to obtain shared memory identifier
  * @param key non-negative integer like fds but system-wide; increases to
  *    maximum, then wraps to 0. Use IPC_PRIVATE to let system choose key.
  * @param size size of share memory segment in bytes
  * @param flag can be file type permission (e.g. 664) ORd with IPC_CREAT
  *     to create memory segment if does not exist
  * @return new key or -1 on error
  */
int shmget(key_t key, int size, int flag);
```

Inter-Process Communication

Shared Memory

- POSIX C library provides APIs managing shared memory.

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>

/**
  * shmat: used to map shared memory to address space
  * @param: shmid the shared memory id returned by shmget
  * @param shmaddr is the address to map it to; 0 allows system to choose
  * @param shmflg has bit fields for additional options; 0 for none
  * @return mapped memory address or (void*)-1 for error
  * @see shmdt(const void* shmaddr) to unmap memory
  */
void *shmat(int shmid, const void *shmataddr, int shmflg);
```

Inter-Process Communication

Example: Shared Memory Time Server/Client

- POSIX C library provides APIs managing shared memory.

```
/* shm_server.c -- shared memory server */
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>

/** time server shared memory buffer length */
static const int MAXSIZE = 128;

/** Shared memory key for time server */
static const key_t TIMESERVER = 5278;
```

Inter-Process Communication

Example: Shared Memory Time Server/Client

- POSIX C library provides APIs managing shared memory.

```
/** This sever responds to a client request through the shared memory buffer
 *   buffer by returning the  current time as a string through the buffer.
 */
int main() {
   // get key for shared memory; create if does not exist
   int shmid = shmget(TIMESERVER, MAXSIZE, IPC_CREAT | 0666);

   if (shmid < 0) {
      perror("shmget");
      exit(1);
   }

   // map shared memory address
   char* shmaddr = shmat(shmid, NULL, 0);
   if (shmaddr == (char *)(-1)) {
      die("shmat");
      exit(1);
   }
```

Inter-Process Communication

Example: Shared Memory Time Server/Client

- POSIX C library provides APIs managing shared memory.

```
// Wait for client to set '?' request to buffer
printf("waiting for request from client\n");
shmaddr[0] = '\0';
while (shmaddr[0] != '?') {
   sleep(1);
}
// format current time
time_t timer;
time(&timer);
struct tm* tm_info = localtime(&timer);
char buf[MAXSIZE];
strftime(buf, 26, "%Y-%m-%d %H:%M:%S", tm_info);
```

Inter-Process Communication

Example: Shared Memory Time Server/Client

- POSIX C library provides APIs managing shared memory.

```
    printf("returning time to client\n");

    // copy backward to ensure '?' is overwritten last
    for (int i = strlen(buf); i >= 0; i--) {
      shmaddr[i] = buf[i];
    }

    // unmap shared memory
    shmdt(shmaddr);

    exit(0);
}
```

Inter-Process Communication

Example: Shared Memory Time Server/Client

- POSIX C library provides APIs managing shared memory.

```
/* shm_client.c -- shared memory client */
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>


/** time server shared memory buffer length */
static const int MAXSIZE = 128;

/** Shared memory key for time server */
static const key_t TIMESERVER = 5278;
```

Inter-Process Communication

Example: Shared Memory Time Server/Client

- POSIX C library provides APIs managing shared memory.

```
/** This sever responds to a client request through the shared memory
 *   buffer by returning the  current time as a string through the buffer.
 */
int main() {
   // get key for shared memory; must already exist
   int shmid = shmget(TIMESERVER, MAXSIZE, 0666);
   if (shmid < 0) {
      perror("shmget");
      exit(1);
   }

   // map shared memory address
   char* shmaddr = shmat(shmid, NULL, 0);
   if (shmaddr == (char *)(-1)) {
      perror("shmat");
      exit(1);
   }
```

Inter-Process Communication

Example: Shared Memory Time Server/Client

- POSIX C library provides APIs managing shared memory.

```
    printf("requesting time from server\n");

    // '?' as first character is request for time
    shmaddr[0] = '?';
    while (shmaddr[0] == '?') {
        sleep(1);
    }

    // display time returned from server
    printf("Time: %s\n", shmaddr);

    // unmap shared memory
    shmdt(shmaddr);

    exit(0);
}
```

Inter-Process Communication

Example: Shared Memory Time Server/Client

- Use *ipcs -m* to view shared memory segments information:

```
IPC status from <running system> as of Tue Aug  8 22:51:29 PDT 2017
T     ID     KEY        MODE        OWNER    GROUP
Shared Memory:
m  65536 0x53414e44 --rw-rw-rw-     phil    staff
m  65537 0x0000162e --rw-rw-rw-     phil    staff
```

Inter-Process Communication

Named Pipes

- A named pipe (also known as a FIFO) is a method for inter-process communication.
  - Extension to the traditional pipe concept on Unix. A traditional pipe is “unnamed” and lasts only as long as the process.
  - Can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used.
  - Usually appears as a special FIFO special file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.

Inter-Process Communication

Named Pipes

- A named pipe (also known as a FIFO) is a method for intern-process communication.
  - Entered into the filesystem by calling mkfifo() in C, or using the mkfifo command line tool, creates i-node file type S_IFIFO.
  - Once created, any process can open it for reading or writing, in the same way as an ordinary file. Must be open at both ends before doing any input or output operations on it.

Inter-Process Communication

Example: Program to Illustrate Named Pipes

```
/** name_pipe_read_first.c – read first, then write. */
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
   int fd1;
   char str1[80], str2[80];

   // FIFO file path
   char * myfifo = "/tmp/myfifo";

   // Creating the named file(FIFO)
   // mkfifo(<pathname>,<permission>)
   mkfifo(myfifo, 0666);
```

Inter-Process Communication

Example: Program to Illustrate Named Pipes

```
    while (1) {
        // First open in read only and read
        printf("User1: ");
        fd1 = open(myfifo,O_RDONLY);
        read(fd1, str1, 80);

        // Print the read string and close
        printf("%s\n", str1);
        close(fd1);

        // Now open in write mode and write
        // string taken from user.
        printf("User2?  ");
        fd1 = open(myfifo,O_WRONLY);
        fgets(str2, 80, stdin);
        write(fd1, str2, strlen(str2)+1);
        close(fd1);
    }
    return 0;
}
```

Inter-Process Communication

Example: Program to Illustrate Named Pipes

```
/** name_pipe_read_write.c – write first, then read. */
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
   int fd1;
   char arr1[80], arr2[80];

   // FIFO file path
   char * myfifo = "/tmp/myfifo";

   // Creating the named file(FIFO)
   // mkfifo(<pathname>,<permission>)
   mkfifo(myfifo, 0666);
```

Inter-Process Communication

Example: Program to Illustrate Named Pipes

```
while (1)  {
   printf("User1?  “);

   // Open FIFO for write only
   fd = open(myfifo, O_WRONLY);

   // Take an input arr2 from user.
   // 80 is maximum length
    fgets(str2, 80, stdin);

   // Write the input arr2 on FIFO and close it
   write(fd, str2, strlen(str2)+1);
   close(fd);
```

Inter-Process Communication

Example: Program to Illustrate Named Pipes

```
        printf("User2: ");

        // Open FIFO for Read only
        fd = open(myfifo, O_RDONLY);

        // Read from FIFO
        read(fd, str1, sizeof(str1));

        // Print the read message
        printf("%s\n", str1);
        close(fd);
    }
    return 0;
}
```

Inter-Process Communication

Remote Inter-Process Communication

- Two means of inter-process communication among processes on different computers:
  - *Socket* is one endpoint of a two-way communication link between two processes running on the network. It is bound to a port number so that the networking layer can identify the process that data will be sent to.
  - *Remote procedure call* (*RPC*) is a protocol that one program can use to request a service from a process running in another computer on a network by calling a procedure of the remote process in a way similar to a local procedure call.

Inter-Process Communication

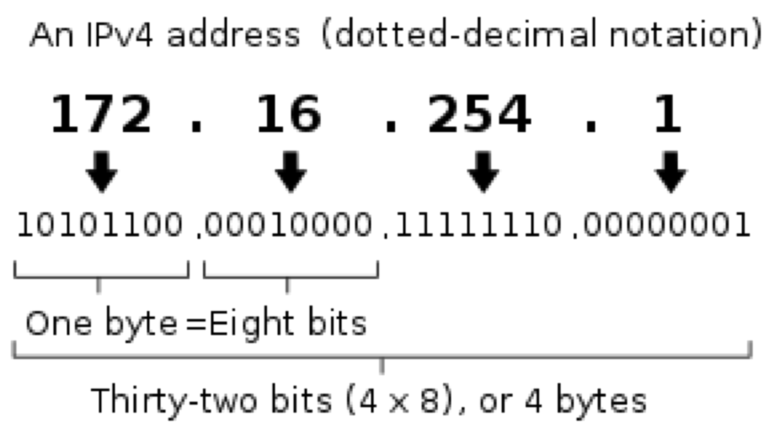
Sockets

- A socket is one endpoint of a two-way communication link between two processes on the network.
- An endpoint is a combination of an *IP address* and a *port number*. Every connection can be uniquely identified by its two endpoints.
- A server runs on a specific computer and listens to the socket for a client running on another server to make a connection request.

Inter-Process Communication

Sockets

- An *IP address* serves two principal functions.
  - It identifies the host, or more specifically its network interface.
  - It provides the location of the host in the network, and thus the capability of establishing a path to that host.



Inter-Process Communication

Sockets

- A network *port* identifies one side of a connection between two computers. As network addresses are like street address, port numbers are like room numbers.
- Computers use port numbers to determine to which process a message should be delivered. An application “listens” on a port for incoming data.
- The process of associating a port number with a service is known as “binding.” The operating systems delivers the data on a port to the process bound to that port.

Inter-Process Communication

Sockets

- Commonly available services like email or a web server use standardized port numbers. Here are some well-known ports for some of these services.

Port number	Process name	Protocol used	Description
20	FTP-DATA	TCP	File transfer—data
21	FTP	TCP	File transfer—control
22	SSH	TCP	Secure Shell
23	TELNET	TCP	Telnet
25	SMTP	TCP	Simple Mail Transfer Protocol
53	DNS	TCP and UDP	Domain Name System
69	TFTP	UDP	Trivial File Transfer Protocol
80	HTTP	TCP and UDP	Hypertext Transfer Protocol
110	POP3	TCP	Post Office Protocol 3
123	NTP	TCP	Network Time Protocol
143	IMAP	TCP	Internet Message Access Protocol
443	HTTPS	TCP	Secure implementation of HTTP

Inter-Process Communication

Sockets

- A client requests a connection using the IP address of the server and the port number on which the server is listening.
- The client identifies itself to the server by binding to a local port number that it uses during this connection (usually assigned by the system).

Inter-Process Communication

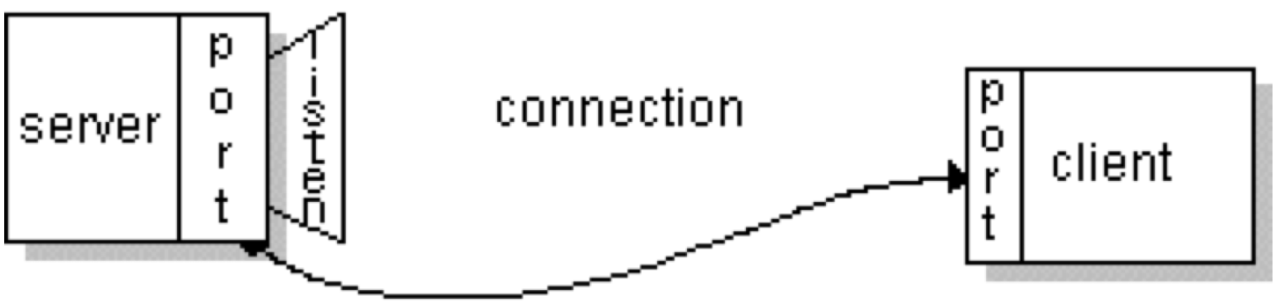
Sockets

- If all goes well, the server accepts the connection.
- Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client.
- It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.

Inter-Process Communication

Sockets

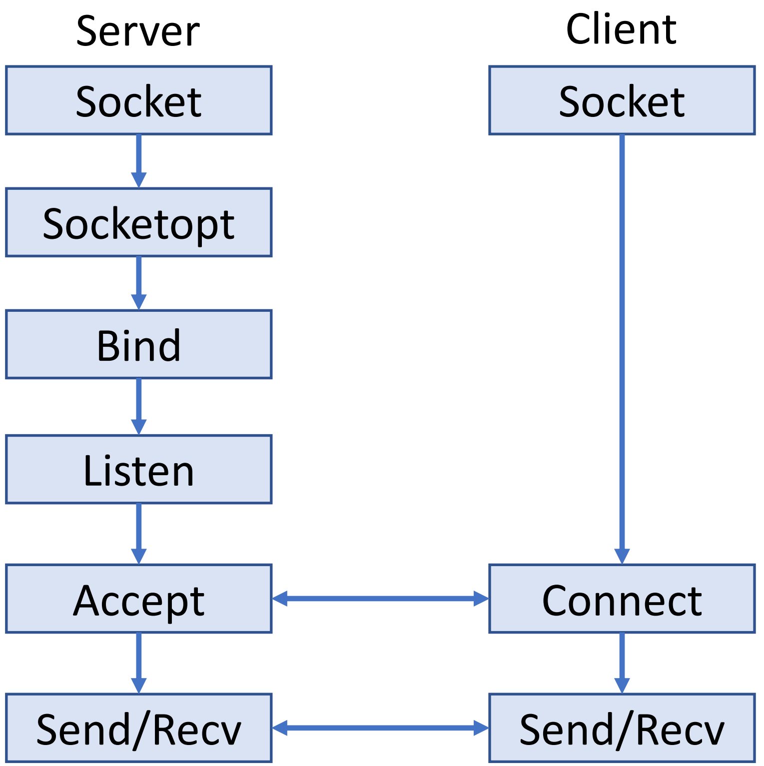
- On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.
- The client and server can now communicate by writing to or reading from their sockets



Inter-Process Communication

Sockets

- Here is a state diagram for server and client



Inter-Process Communication

Example: Simple DateServer and DateClient

```
#import  <stdbool.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define PORT 9091
#define MAXBUF 128

int main() {
    int listen_sock_fd = newListenSocket(PORT);
    if (listen_sock_fd == 0) {
        perror("newListenSocket");
        exit(EXIT_FAILURE);
    }
    fprintf(stderr, "listen: port %d\n", PORT);
```

Inter-Process Communication

Example: Simple DateServer and DateClient

```
while (true) {
         // accept client connection
        int peer_sock_fd = acceptPeerConnection(listen_sock_fd);
        fprintf(stderr, "connection?\n");
        if (peer_sock_fd < 0) {
                perror("accept");
                continue;
        }

        char host[MAXBUF];
        int port;
        if (getLocalHostAndPort(peer_sock_fd, host, &port) == 0) {
                fprintf(stderr, "  local: %s:%d\n", host, port);  // show local host:port
        }

        if (getPeerHostAndPort(peer_sock_fd, host, &port) == 0) {
                fprintf(stderr, "  peer:  %s:%d\n", host, port);  // show peer host:port
        }
```

Inter-Process Communication

Example: Simple DateServer and DateClient

```
            // format current time
            time_t timer;
            time(&timer);
            struct tm* tm_info = localtime(&timer);
            char buf[MAXBUF];
            strftime(buf, MAXBUF, "%Y-%m-%d %H:%M:%S", tm_info);

            // send time
            if (write(peer_sock_fd , buf , strlen(buf)+1) < 0) {
                perror("send");
            }
            fprintf(stderr, "  sent:  %s\n", buf);

            // close socket to complete write operation
            close(peer_sock_fd);
    }
    // close server socket
    close(listen_sock_fd);
    return 0;
}
```

Inter-Process Communication

Example: Simple DateServer and DateClient

```
            // send time
            if (write(socket_fd , buf , strlen(buf)+1) < 0) {
                perror("send");
            }
            // close socket to complete write operation
            close(socket_fd);
        }

        // close listener socket
        close(listen_sock_fd);
        return 0;
    }
```

Inter-Process Communication

Example: Simple DateServer and DateClient

```
/**
 * Create new listen socket on specified port
 *
 * @param port the port
 * @return listen socket or 0 if creation, bind, or listen failed
 */
int newListenSocket(int port) {
   // Creating internet socket stream file descriptor
   int listen_sock_fd = socket(AF_INET, SOCK_STREAM, 0);
   if (listen_sock_fd == 0) {
      return 0;
   }

   // SO_REUSEADDR prevents the "address already in use" errors when testing servers.
   int optval = 1;
   if (setsockopt(listen_sock_fd, SOL_SOCKET, SO_REUSEADDR, &optval , sizeof(int)) < 0) {
      close(listen_sock_fd);
      return 0;
   }
```

Inter-Process Communication

Example: Simple DateServer and DateClient

```
// listener address and port
struct sockaddr_in listen_addr;
socklen_t listen_size = sizeof(listen_addr);
memset(&listen_addr, 0, listen_size);
listen_addr.sin_family = AF_INET;  // address from internet
listen_addr.sin_port = htons(PORT);   // port in network byte order
listen_addr.sin_addr.s_addr = INADDR_ANY;  // bind to any address

// bind host address and port
if (bind(listen_sock_fd, (struct sockaddr *)&listen_addr, listen_size) < 0) {
        close(listen_sock_fd);
        return 0;
}
```

Inter-Process Communication

Example: Simple DateServer and DateClient

```
        // set up queue for clients connections up to default
        // maximum pending socket connections (usually 128)
        if (listen(listen_sock_fd, SOMAXCONN) < 0) {
                close(listen_sock_fd);
                return 0;
        }

    return listen_sock_fd;
}
```

Inter-Process Communication

Example: Simple DateServer and DateClient

```
/**
 * Accept new peer connection on a listen socket.
 *
 * @param listen_sock_fd the listen socket
 * @return the peer socket fd
 */
int acceptPeerConnection(int listen_sock_fd) {
    while (true) {
        struct sockaddr_in peer_addr;
        socklen_t peer_size = sizeof(peer_addr);
        int peer_sock_fd =
            accept(listen_sock_fd, (struct sockaddr *)&peer_addr, &peer_size);
        if (peer_sock_fd > 0) {
            return peer_sock_fd;
        }
    }
}
```

Inter-Process Communication

Example: Simple DateServer and DateClient

```
/**
 * Get the local host and port for a socket.
 *
 * @param sock_fd the socket
 * @param addr_str buffer for IP address string
 * @param port pointer for port value
 * @return 0 if successful
*/
int getLocalHostAndPort(int sock_fd, char *addr_str, int *port) {
     struct sockaddr_in addr;
     socklen_t size = sizeof(addr);

     int status = getsockname(sock_fd, (struct sockaddr *)&addr, &size);
   if (status == 0) {
     *port = ntohs(addr.sin_port);
     strcpy(addr_str, inet_ntoa(addr.sin_addr));
   }
   return status;
}
```

Inter-Process Communication

Example: Simple DateServer and DateClient

```
/**
 * Get the peer host and port for a socket.
 *
 * @param sock_fd the socket
 * @param addr_str buffer for IP address string
 * @param port pointer for port value
 * @return 0 if successful
 */
int getPeerHostAndPort(int sock_fd, char *addr_str, int *port) {
    struct sockaddr_in addr;
    socklen_t size = sizeof(addr);

    int status = getpeername(sock_fd, (struct sockaddr *)&addr, &size);
    if (status == 0) {
        *port = ntohs(addr.sin_port);
        strcpy(addr_str, inet_ntoa(addr.sin_addr));
    }
    return status;
}
```

Inter-Process Communication

Example: Simple DateServer and DateClient

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>

#define PORT 9091
#define MAXBUF 128

int main(int argc, char const *argv[]) {
    printf("Enter IP Address of the date service on port %d:", PORT);
    char host_str[MAXBUF];
    if (fgets(host_str, MAXBUF, stdin) == NULL) {
        perror("fgets");
        return EXIT_FAILURE;
    }
```

Inter-Process Communication

Example: Simple DateServer and DateClient

```
// trim newline
size_t len = strlen(host_str);
if (len > 0 && host_str[len-1] == '\n') host_str[len-1] = 0;
else if (len > 1 && host_str[len-2] == '\r') host_str[len-2] = 0;

// connect to server host and port
int server_sock_fd = connectPeerHostAndPort(host_str, PORT);

char host[MAXBUF];
int port;
if (getLocalHostAndPort(server_sock_fd, host, &port) == 0) {
    fprintf(stderr, "  local: %s:%d\n", host, port); // show local host:port
}

if (getPeerHostAndPort(server_sock_fd, host, &port) == 0) {
    fprintf(stderr, "  peer:  %s:%d\n", host, port); // show peer host:port
}
```

Inter-Process Communication

Example: Simple DateServer and DateClient

```
        char buf[MAXBUF];
        if (read(server_sock_fd , buf, MAXBUF) < 0) {
                perror("read");
        } else {
                fprintf(stderr, "  time:  %s\n", buf);
        }

        close(server_sock_fd);
        return EXIT_SUCCESS;
}
```

Inter-Process Communication

Example: Simple DateServer and DateClient

```
/**
 * Connect to peer at host and port.
 *
 * @param host the host IP address as a string
 * @param port the host port
 * @return the peer socket fd
 */
int connectPeerHostAndPort(const char *host, int port) {
    int sock_fd = 0;
    if ((sock_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        return -1;
    }

    struct sockaddr_in host_addr;
    const socklen_t addrlen = sizeof(host_addr);
    memset(&host_addr, 0, addrlen);
    host_addr.sin_family = AF_INET;  // address from internet
    host_addr.sin_port = htons(port);   // port in network byte order
```

Inter-Process Communication

Example: Simple DateServer and DateClient

```
        // Convert IPv4 and IPv6 addresses from text to binary form
        if (inet_pton(AF_INET, host, &host_addr.sin_addr) <= 0) {
                close(sock_fd);
                return -1;
        }

        // connect to server
        if (connect(sock_fd, (struct sockaddr *)&host_addr, addrlen) != 0) {
                close(sock_fd);
                return -1;
        }

        return sock_fd;
}
```