Network Programming

TCP Client-Server view
Connection-oriented socket connections

Server Side Socket Details

- Create socket
- Bind to a port
- listen for incoming connections
- accept an incoming connection
- connect to server's socket
- read from the connection
- write to the connection
- close connection

#include  <stdio.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <netinet/in.h>
#include  <netdb.h>
#define   PORTNUM  8824
#define   oops(msg)      { perror(msg) ; exit(1) ; }

Example: A Simple Time Server

#include  <stdio.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <netdb.h>
#define   PORTNUM  8824
#define   max_serv    10
#define   serv_addr  ( struct sockaddr_in   { struct sockaddr         { } })

#include  <netinet/in.h>
#include  <iostream.h>
#include  <stdlib.h>
#include  <string.h>

int main(int argc, char *argv[])
{
    int sockfd, portno, n;
    char *server, *buffer, *host, *port;
    socklen_t clilen;
    struct sockaddr_in serv_addr;
    struct hostent *server;
    ....
    close(sockfd);
    exit(0);
}

int main(int argc, char *argv[])
{
    int sockfd, portno, n;
    char *server, *buffer, *host, *port;
    socklen_t clilen;
    struct sockaddr_in serv_addr;
    struct hostent *server;
    ....
    close(sockfd);
    exit(0);
}
void main(int ac, char **av)
{
  struct sockaddr_in saddr; /* build our address here */
  /* build hostent */
  struct hostent *hp; /* this is part of our */
  char hostname[256]; /* address */
  int slen,sock_id,sock_fd; /* line id, file desc */
  FILE *sock_fp; /* use socket as stream */
  char *ctime(); /* convert secs to string */
  long time(), thetime; /* time and the val */

  gethostname(hostname, 256); /* where am I? */
  hp = gethostbyname(hostname); /* get info about host */
  bzero(&saddr, sizeof(saddr)); /* zero struct */
  /* fill in hostaddr */
  bcopy(hp->h_addr, &saddr.sin_addr, hp->h_length);
  saddr.sin_family = AF_INET; /* fill in socket type */
  saddr.sin_port = htons(PORTNUM); /* fill in socket port */

  sock_id = socket(AF_INET, SOCK_STREAM, 0); /* get a socket */
  if (sock_id == -1) oops( "socket" );

  if (bind(sock_id, &saddr, sizeof(saddr)) != 0)/* bind it to */
    oops("bind"); /* an address */

  if (listen(sock_id, 1) != 0) oops("listen");

  while (1) { /* read the line */
    sock_fd = accept(sock_id, NULL, NULL); /* wait for call */
    printf("** Server: A new client connected!\n");
    if (sock_fd == -1)
      oops("accept"); /* error getting calls */

    sock_fp = fdopen(sock_fd,"w"); /* we'll write to the */
    if (sock_fp == NULL)
      oops("fdopen"); /* unless we can't */

    thetime = time(NULL); /* get time */
    /* and convert to strng */
    fprintf(sock_fp, "**************************************\n");
    fprintf(sock_fp, "** From Server: The current time is: \n");
    fprintf(sock_fp, "\n%" , ctime(&thetime));
    fprintf(sock_fp, "** From Server: The current time is: \n");
    fprintf(sock_fp, "**************************************\n");

    fclose(sock_fp); /* release connection */

    fflush(stdout); /* force output */
  }
}

main(int argc, char **argv){
  int len, port_sk,  client_sk;
  char *errmess;

  port_sk = tcp_passive_open(port); /* establish port */
  if (port_sk < 0) { perror("socket"); exit(1); }
  printf("start up complete\n");

  client_sk = tcp_accept(port_sk); /* wait for client to connect */
  close(port_sk); /* only want one client, so close port_sk */

  for(;;) { /* talk to client */
    len = read(client_sk,buff,buf_len);  //listen
    printf("client says: %s\n",buff);
    ....
    if ( gets(buff) == NULL ) { /* user typed end of file */
      close(client_sk); break;
    }
    write(client_sk,buff,strlen(buff));    //server
  }
  exit(0);
}

int tcp_passive_open(portno){
  int portno;

  { int     sd, code;
    struct sockaddr_in bind_addr;
    bind_addr.sin_family = AF_INET;
    bind_addr.sin_addr.s_addr = 0;    /* 0.0.0.0 == this host */
    bzero(&bind_addr.sin_zero, 8);
    bind_addr.sin_port = htons(portno);
    sd = socket(AF_INET, SOCK_STREAM,0);
    if ( sd < 0 ) return sd;
    code = bind(sd, &bind_addr, sizeof(bind_addr));
    if ( code < 0 ) { close(sd); return code; }
    code = listen(sd,1);
    if ( code < 0 ) { close(sd); return code; }
  return sd;
  }
}

int tcp_accept(sock){
  int sock;

  { int ad;
    struct sockaddr bind_addr;
    int leaveafter(sizeof(kind_addr));
    sd = accept(sock, kind_addr, &ad);
    return ad;
  }
}
main ( int argc, char**argv )
!
int serv_sk, len;
char *errmess;
!
serv_sk = tcp_active_open(host, port); /* request connection */
if ( serv_sk < 0 ) { perror("socket"); exit(1); }
printf("You can send now
");
!
for(;;) { /* talk to server */
if ( gets(buff) == NULL ) { /* client's turn */
close(serv_sk); break;
}
write(serv_sk, buff, strlen(buff));
!
len = read(serv_sk, buff, buf_len); // wait for server's response
if (len == 0) {
printf("server finished the conversation\n\n");break;
}
buff[len] = '\0';
printf("server says: %s\n", buff);
}
exit(0);
!
!
!
!
!
!
!
!
!
!
!
!
!

2. client code

active open

int tcp_active_open(char* hostname, int portno)
!
{
int     sd, code;
struct  sockaddr_in bind_addr;
struct hostent *host;

host = gethostbyname(hostname);
if (host == NULL ) return -1;
bind_addr.sin_family = PF_INET;
bind_addr.sin_addr = *((struct in_addr *) (host->h_addr));
bind_addr.sin_port = port;
sd = socket(AF_INET, SOCK_STREAM, 0);
if ( sd < 0 ) return sd;

code = connect(sd, &bind_addr, sizeof(bind_addr) );
if ( code < 0 ) { close(sd); return code; }
return sd;

!
!
!
!
!
!
!
!
!
!
!
!
!

Threads

• In certain cases, a single application may need to run several tasks at the same time
  • Creating a new process for each task is time consuming
  • Use a single process with multiple threads
    • Faster
    • Less overhead for creation, switching, and termination
    • Share the same address space

Thread Creation

• pthread_create
  // creates a new thread executing start_routine
  int pthread_create(pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine)(void*), void *
    "arg");

• pthread_join
  // suspends execution of the calling thread until the target
  // thread terminates
  int pthread_join(pthread_t thread, void **value_ptr);

Thread Example

main();

pthread_t thread1, thread2; /* thread variables */

pthread_create(thread1, NULL, print_message_function,(void*)"hello ");

pthread_create(thread2, NULL, print_message_function,(void*)"world ");

pthread_join(thread1, NULL);

pthread_join(thread2, NULL);

printf("\n");
exit(0);

Why use pthread_join?
To force main block to wait for both threads to terminate, before it exits.
If main block exits, both threads exit, even if the threads have not finished their work.

Thread Example (cont.)

main();

pthread_t thread1, thread2; /* thread variables */

pthread_create(thread1, NULL, print_message_function,(void*)"hello ");

pthread_create(thread2, NULL, print_message_function,(void*)"world ");

pthread_join(thread1, NULL);

pthread_join(thread2, NULL);

printf("\n");
exit(0);
Example: Interthread Cooperation

```c
void* print_count ( void *ptr )
{
  int i;
  for (i=0;i<NUM;i++){
    printf("counter = %d \n", counter);
    //sleep(1);
  }
  pthread_exit(0);
}

void* increment_count ( void *ptr )
{
  int i;
  for (i=0;i<NUM;i++){
    counter++;
    //sleep(1);
  }
  pthread_exit(0);
}
```

Interthread Cooperation (cont.)

```c
void* print_count ( void *ptr )
{
  int i;
  for (i=0;i<NUM;i++){
    printf("counter = %d \n", counter);
    //sleep(1);
  }
  pthread_exit(0);
}

void* increment_count ( void *ptr )
{
  int i;
  for (i=0;i<NUM;i++){
    counter++;
    //sleep(1);
  }
  pthread_exit(0);
}
```

Concurrent Issues

- If programs are independent, the results are the same (X=1)
- If programs are executed concurrently and one program is X=1, are results of P1 and P2 different
- "interleaving" makes it difficult to deal with global properties from the local analysis!
- assumption: access to the memory is atomic

POSIX Threads: MUTEX

```c
#include <pthread.h>

... 

pthread_mutex_t my_mutex;
// should be of global scope
...

int main()
{
  int tmp;
  ...
  // initialize the mutex
  tmp = pthread_mutex_init( &my_mutex, NULL );
  ...
  // create threads
  ...
  pthread_mutex_lock( &my_mutex );
  do_something_private();
  pthread_mutex_unlock( &my_mutex );
  ...
  return 0;
}
```

Whenever a thread reaches the lock/unlock block, it first determines if the mutex is locked. If no, it waits until it is unlocked. Otherwise, it takes the mutex, does the succeeding code, then frees the mutex and unlocks the code when it's done.

MUTEX Example

```c
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

* a new data type named `pthread_mutex_t` is designated for mutexes
* a mutex is a key (to access the code section) that is handed over to only one thread at a time
* the attribute of a mutex can be controlled by using the `pthread_mutex_init()` function
* the lock/unlock functions work in tandem

Concurrency Issues

- Shared variables are an effective way to communicate between processes
  - X:=X-1 is implemented as 3 different instructions
    - load the value of X to the register
    - increment the register
    - store the value of register to X
- Two processes updating same variable concurrently causes erroneous results
- Correctness of the program needs that this updating will be indivisible (or atomic)
  - Reading a variable can also be a critical section
    - e.g. reading four bytes that are not volatile
POSIX: Semaphores

- creating a semaphore:
  ```c
  int sem_init(
      sem_t *sem,
      int pshared,
      unsigned int value);
  ```

  initializes a semaphore object pointed to by sem
  pshared is a sharing option; a value of 0 means the semaphore is
  local to the calling process
  gives an initial value value to the semaphore

- terminating a semaphore:
  ```c
  int sem_destroy(
      sem_t *sem);
  ```

  frees the resources allocated to the semaphore sem
  usually called after pthread_join()
  an error will occur if a semaphore is destroyed for which a thread

POSIX: Semaphores (cont.)

- semaphore control:
  ```c
  int sem_post(
      sem_t *sem);
  ```

  sem_post atomically increases the value of a semaphore by 1, i.e.,
  when 2 threads call sem_post simultaneously, the semaphore's
  value will also be increased by 2 (there are 2 atoms calling)

  ```c
  int sem_wait(
      sem_t *sem);
  ```

  sem_wait atomically decreases the value of a semaphore by 1; but
  always waits until the semaphore has a non-zero value first

Semaphore: Example

```c
#include <semaphore.h>

void *thread_function( void *arg );
... 
sem_t semaphore;        // also a global variable just like mutexes 
... 
int main()
{
    ... 
    // initialize the semaphore
    tmp = sem_init( &semaphore, 0, 0 );
    ... 
    // create threads
    pthread_create( &thread[i], NULL, thread_function, NULL );
    ...
    while ( still_has_something_to_do() )
    {
        sem_post( &semaphore );
        ...
    }
    ...
    pthread_join( thread[i], NULL );
    sem_destroy( &semaphore );
    return 0;
}
... 
```

Semaphore: Example (cont.)

```c
void *thread_function( void *arg )
{
    ...
    sem_wait( &semaphore );
    perform_task_when_sem_open();
    ...
    pthread_exit( NULL );
}
```

Exercises

Threads (True or False Questions):

- A thread cannot see the variables on another thread's stack.
  - False -- they can since they share memory

- In a non-preemptive thread system, you do not have to worry about race conditions.
  - False -- as threads block and unblock, they may do so in unspecified orders, so you
    can still have race race conditions.

- A thread may only call pthread_join() on threads which it has created with
  pthread_create()
  - False -- any thread can join with any other

- With mutexes, you may have a thread execute instructions atomically with respect to
  other threads that lock the mutex.
  - True -- That's most often how mutexes are used.

- pthread_create() always returns to the caller
  - True.

- pthread_mutex_lock() never returns
  - False -- It may block, but it when it unblocks, it will return.

- pthread_exit() returns if there is no error
  - False -- never returns.
Exercises

Processes:

Please provide two reasons on why an invocation to fork() might fail:

1. too many processes in the system
2. the total number of processes for the real uid exceeds the limit
3. too many PID in the system
4. memory exceeds the limit

When a process terminates, what would be the PID of its child processes? Why?

It would become 1. Because when any of the child processes terminate, init would be informed and fetch termination status of the process so that the system is not clogged by zombie processes.

Daemon Processes

Daemon Characteristics

Commonly, daemon processes are created to offer a specific service.

- fork off the parent process
- change file mode mask (umask)
- create a unique Session ID (SID)
- change the current working directory to a safe place
- close (or redirect) standard file descriptors
- open any logs for writing
- enter actual daemon code

Example Daemon Creation

```c
int daemon_init(void)
{
    pid_t pid;
    if ((pid=fork())<0) return (-1);
    else if (pid!=0) exit (0); //parent goes away
    setsid(); //becomes session leader
    chdir("/"); //cwd
    umask(0); //clear file creation mask
    return (0)
}
```

Daemon Logging

A daemon cannot simply print error messages to the terminal or standard error. Also, we would not want each daemon writing their error messages into separate files in different formats. A central logging facility is needed.

There are three ways to generate log messages:

- via the kernel routine log(9)
- via the userland routine syslog(3)
- via UDP messages to port 514
Syslog()

syslog(3) allows us to set specific options when logging:
- prepend iden to each message
- specify logging options (LOG_CONS | LOG_NODEST | LOG_PERM | LOG_PID)
- specify a facility (such as LOG_DAEMON, LOG_MAIL, etc.)

syslog(3) writes a message to the system message logger, tagged with priority.
A priority is a combination of a facility (as above) and a level (such as LOG_EMERG, LOG_ALERT or LOG_ERROR).