BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI (RAJASTHAN)
IS F462 – Network Programming
Lab#8

Topics: IO Multiplexing, Pthreads

Writing a concurrent TCP Server with select:

Consider the following code for a simple echo server using select() system call. It takes port number as the input over which it listens for new connections.

```
/*TCPServerwithSelect.c*/
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <unistd.h>
#include <strings.h>
#include <sys/select.h>

#define LISTENQ 15
#define MAXLINE 80

int
main (int argc, char **argv)
{
  int i, maxi, maxfd, listenfd, connfd, sockfd;
  int nready, client[FD_SETSIZE];
  ssize_t n;
  fd_set rset, allset;
  char buf[MAXLINE];
  socklen_t clilen;
  struct sockaddr_in cliaddr, servaddr;

  listenfd = socket (AF_INET, SOCK_STREAM, 0);

  bzero (&servaddr, sizeof (servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
  servaddr.sin_port = htons (atoi(argv[1]));

bind (listenfd, (struct sockaddr *) & servaddr, sizeof (servaddr));

  listen (listenfd, LISTENQ);

  maxfd = listenfd;         /* initialize */
  maxi = -1;                /* index into client[] array */
  for (i = 0; i < FD_SETSIZE; i++)
    client[i] = -1;         /* -1 indicates available entry */
  FD_ZERO (&allset);
  FD_SET (listenfd, &allset);

  for (;;)
    {
      rset = allset;        /* structure assignment */
      nready = select (maxfd + 1, &rset, NULL, NULL, NULL);

      if (FD_ISSET (listenfd, &rset))
       {                    /* new client connection */
         clilen = sizeof (cliaddr);
         connfd = accept (listenfd, (struct sockaddr *) & cliaddr, &clilen);
```

```
        printf ("new client: %s, port %d\n",
                inet_ntop (AF_INET, &cliaddr.sin_addr, 4, NULL),
                ntohs (cliaddr.sin_port));

        for (i = 0; i < FD_SETSIZE; i++)
          if (client[i] < 0)
            {
             client[i] = connfd;  /* save descriptor */
             break;
            }
        if (i == FD_SETSIZE){

          printf ("too many clients");
             exit(0);
       }
          FD_SET (connfd, &allset);/* add new descriptor to set */
          if (connfd > maxfd)
            maxfd = connfd;  /* for select */
          if (i > maxi)
            maxi = i;        /* max index in client[] array */

          if (--nready <= 0)
            continue;        /* no more readable descriptors */
       }

      for (i = 0; i <= maxi; i++)
       {                     /* check all clients for data */
          if ((sockfd = client[i]) < 0)
            continue;
          if (FD_ISSET (sockfd, &rset))
            {
              if ((n = read (sockfd, buf, MAXLINE)) == 0)
               {
                 /*connection closed by client */
                 close (sockfd);
                 FD_CLR (sockfd, &allset);
                 client[i] = -1;
               }
              else
               write (sockfd, buf, n);

              if (--nready <= 0)
               break;        /* no more readable descriptors */
            }
       }
    }
}
```

Q?

1. Go through the above code. See if you can understand how it works. Remember the data structures discussed in the class?
2. Compile the above program and run it with a port number. Connect to this server using a telnet command. `telnet hostip portno`. Observe the output.
3. Connect to the server multiple times using multiple instances of telnet and see if it is operating concurrently. To feel the difference, let the server wait for some seconds before replying.
4. Modify the above server such that server accepts only MAX_CLIENT number of connections. If a new connection arrives and the limit is already reached, how do

you handle it?

5. This is just a simple echo server. It is stateless protocol. Suppose, the server has to reply cumulatively, that is server accumulates all the strings it has received so far and replies all that with the new string sent by the client. This is stateful protocol. Make the modifications to the server to behave in this way.
6. Servers with select have the loophole of denial of service attack. Suppose the protocol is that server has to wait until it receives a minimum of 3 characters from the client which specify the length of the string the client will send. Modify the server for this case. In telnet send only two characters, and don't send the third char. Observe the servers response at this time by trying to connect from another telnet.
7. Modify your server to withstand DOS attacks as mentioned in 6.

Threads programming:

The following programs provide multi threaded echo client and servers.

```
/*threadclient.c*/
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <unistd.h>
#include <strings.h>
#include <string.h>
#include <pthread.h>

void   *copyto(void *);

static int    sockfd;             /* global for both threads to access */
static FILE   *fp;

void
str_cli(FILE *fp_arg, int sockfd_arg)
{
       char          recvline[1000];
       pthread_t     tid;

       sockfd = sockfd_arg; /* copy arguments to externals */
       fp = fp_arg;
       pthread_create(&tid, NULL, copyto, NULL);

       while (read(sockfd, recvline, 1000) > 0)
              fputs(recvline, stdout);
}

void *copyto(void *arg)
{
       char   sendline[1000];
        while (fgets(sendline, 1000, fp) != NULL)
              write(sockfd, sendline, strlen(sendline));

       shutdown(sockfd, SHUT_WR); /* EOF on stdin, send FIN */

       return(NULL);
}


int main(int argc,char **argv)
```

```
{
	int sockfd;
	struct sockaddr_in servaddr;
	sockfd=socket(AF_INET,SOCK_STREAM,0);
	bzero(&servaddr,sizeof(servaddr));
	servaddr.sin_family=AF_INET;
	servaddr.sin_port=htons(atoi(argv[1]));
	servaddr.sin_addr.s_addr=inet_addr("172.24.2.4");
	if((connect(sockfd,(struct sockaddr *)&servaddr,sizeof(servaddr)))<0)
	printf("connect error\n");
	str_cli(stdin,sockfd);
	exit(0);
}
```

/*threadeserver.c*/

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

void str_echo(int sockfd)
{
	ssize_t n;
	char buf[1000];
	while((n=read(sockfd,buf,1000))>0)
	{
	write(sockfd,buf,n);
	printf("%s\n",buf);
	}
}

static void   *doit(void *);             /* each thread executes this function */

int
main(int argc, char **argv)
{
	int                         listenfd, connfd;
	socklen_t           addrlen, len;
	pthread_t tid1;
	struct sockaddr     *cliaddr;
	struct sockaddr_in servaddr;
	listenfd=socket(AF_INET,SOCK_STREAM,0);
	bzero(&servaddr,sizeof(servaddr));
	servaddr.sin_family=AF_INET;
	servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
	servaddr.sin_port=htons(atoi(argv[1]));
	bind(listenfd,(struct sockaddr *)&servaddr,sizeof(servaddr));
	listen(listenfd,10);

	cliaddr = malloc(addrlen);

	for ( ; ; ) {
		len = addrlen;
		connfd = accept(listenfd, cliaddr, &len);
		pthread_create(&tid1, NULL, &doit, (void *) connfd);
	}
}

static void *doit(void *arg)
```

```
{
        pthread_detach(pthread_self());
        str_echo((int) arg); /* same function as before */
        close((int) arg);          /* we are done with connected socket */
        return(NULL);
}
```

Q?

1. Go through the code and try to understand how it is working
2. Compile the above programs using the flag –lpthread. Try to connect using multiple clients. You can see the threads using ps –eLf command. Under NLWP heading is the number of threads in that process.
3. Extend the above program for the following
   Maintain the number of threads not more than MAX_THREADS.
   a. Case1: If a new request comes but the limit is reached send RST segment.
   b. Case2: if a new request comes, don't accept unless threads have got released.
4. Consider pre-threaded server. The control thread creates n threads and lets each thread call accept(). Following code will help in implementing a pre-threaded server.

```
/*in control thread*/
for (i = 0;  i < nthreads; i++)
         thread_make(i);          /* only main thread returns */

 void thread_make(int i)
 {
 void     *thread_main(void *);
 Pthread_create(&tptr[i].thread_tid, NULL, &thread_main, (void *) i);
   return;                     /* main thread returns */
 }

void * thread_main(void *arg)
 {
     int     connfd;
     void    web_child(int);
     socklen_t clilen;
     struct sockaddr *cliaddr;

     cliaddr = malloc(addrlen);

     printf("thread %d starting\n", (int) arg);
     for ( ; ; ) {
         clilen = addrlen;
         Pthread_mutex_lock(&mlock);
         connfd = accept(listenfd, cliaddr, &clilen);
         Pthread_mutex_unlock(&mlock);
         tptr[(int) arg].thread_count++;

         web_child(connfd);      /* process request */
         Close(connfd);
     }
 }
```

The following is the demonstration of use of condition variable in threads.

/*condvar.c*/

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define checkResults(string, val) {             \
 if (val) {                                     \
   printf("Failed with %d at %s", val, string); \
   exit(1);                                     \
 }                                              \
}

#define                 NUMTHREADS    10
pthread_mutex_t         dataMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t          dataPresentCondition = PTHREAD_COND_INITIALIZER;
int                     dataPresent=0;
int                     sharedData=0;

void *theThread(void *parm)
{
   int   rc;
   int   retries=2;

   printf("Consumer Thread %.8x %.8x: Entered\n", pthread_self());
   rc = pthread_mutex_lock(&dataMutex);
   checkResults("pthread_mutex_lock()\n", rc);

   while (retries--) {
      /* The boolean dataPresent value is required for safe use of */
      /* condition variables. If no data is present we wait, other */
      /* wise we process immediately.                              */
      while (!dataPresent) {
         printf("Consumer Thread %.8x %.8x: Wait for data to be produced\n");
         rc = pthread_cond_wait(&dataPresentCondition, &dataMutex);
         if (rc) {
            printf("Consumer Thread %.8x %.8x: condwait failed, rc=%d\n",rc);
            pthread_mutex_unlock(&dataMutex);
            exit(1);
         }
      }
      printf("Consumer Thread %.8x %.8x: Found data or Notified, "
             "CONSUME IT while holding lock\n",
             pthread_self());
      /* Typically an application should remove the data from being    */
      /* in the shared structure or Queue, then unlock. Processing     */
      /* of the data does not necessarily require that the lock is held */
      /* Access to shared data goes here */
      --sharedData;
      /* We consumed the last of the data */
      if (sharedData==0) {dataPresent=0;}
      /* Repeat holding the lock. pthread_cond_wait releases it atomically */
   }
   printf("Consumer Thread %.8x %.8x: All done\n",pthread_self());
   rc = pthread_mutex_unlock(&dataMutex);
   checkResults("pthread_mutex_unlock()\n", rc);
   return NULL;
}

int main(int argc, char **argv)
{
   pthread_t             thread[NUMTHREADS];
   int                   rc=0;
```

```
   int                  amountOfData=1000;
   int                  i;

   printf("Enter Testcase - %s\n", argv[0]);

   printf("Create/start threads\n");
   for (i=0; i <NUMTHREADS; ++i) {
 rc = pthread_create(&thread[i], NULL, theThread, NULL);
      checkResults("pthread_create()\n", rc);
   }

   /* The producer loop */
   while (amountOfData--) {
      printf("Producer: 'Finding' data\n");
      sleep(3);

      rc = pthread_mutex_lock(&dataMutex);   /* Protect shared data and flag
*/
      checkResults("pthread_mutex_lock()\n", rc);
      printf("Producer: Make data shared and notify consumer\n");
      ++sharedData;                          /* Add data
*/
      dataPresent=1;                         /* Set boolean predicate
*/

      rc = pthread_cond_signal(&dataPresentCondition); /* wake up a consumer
*/
      if (rc) {
         pthread_mutex_unlock(&dataMutex);
         printf("Producer: Failed to wake up consumer, rc=%d\n", rc);
         exit(1);
      }

      printf("Producer: Unlock shared data and flag\n");
      rc = pthread_mutex_unlock(&dataMutex);
      checkResults("pthread_mutex_lock()\n",rc);
   }

   printf("Wait for the threads to complete, and release their resources\n");
   for (i=0; i <NUMTHREADS; ++i) {
  rc = pthread_join(thread[i], NULL);
      checkResults("pthread_join()\n", rc);
   }

   printf("Clean up\n");
   rc = pthread_mutex_destroy(&dataMutex);
   rc = pthread_cond_destroy(&dataPresentCondition);
   printf("Main completed\n");
   return 0;
}
```

Q?

1. Compile it with -`lpthread` option.
2. Observe that producer is signaling and consumer is waiting for the event. Will it be possible the other way?
3. What is the problem if `dataPresentCondition` variable is not used?
4. Remove mutex protection and see what happens?
5. There are 10 consumer threads and 1000 data items. Each consumer consumes only one data item. Modify the above program to do like this: each consumer consumes as many

objects as possible until producer signals them that it has completed producing all items.

===End of Lab8===