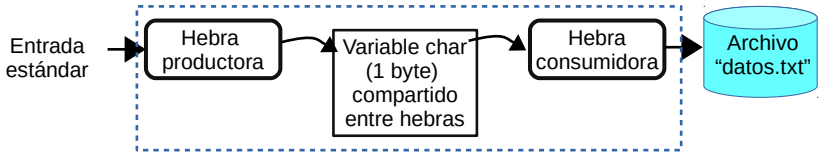
Segundo Certamen
Tiempo 90 min. Responder un problema por página

1.- 50 Puntos

Se tiene un programa que lee datos de la entrada estándar y los almacena en un archivo, ver http://profesores.elo.utfsm.cl/~agv/elo330/2s18/C2/p1.c

Cuando llega un gran flujo de datos por la entrada estándar se piensa que la escritura en el archivo (disco) puede retardar todo el proceso. Se propone efectuar la escritura en disco en una hebra separada.

Modifique el programa para que una hebra atienda la lectura de un byte a la vez desde la entrada estándar y otra hebra se haga cargo de la escritura en disco.



Ayuda: Además del uso de un mutex para acceder a la variable compartida (byte), considere el uso de dos variables de condición: una para que la hebra consumidora espere hasta que el byte sea escrito y otra para que la hebra productora espere hasta que el byte sea leído.

```
/* Para compilar:
$ gcc -o p1 -pthread p1_sol.c

No es obvio ni esperable que aquí se use un while en lugar de un if
en la sentencia while(!byteFull) y while (byteFull).
Esto se explica por la documentación man para esta función:
When using condition variables there is always a Boolean predicate involving shared variables associated with each
condition wait that is true if the thread should proceed. Spurious wakeups from the pthread_cond_timedwait() or
pthread_cond_wait() functions may occur. Since the return from pthread_cond_timedwait() or pthread_cond_wait()
does not imply anything about the value of this predicate, the predicate should be re-evaluated upon such return."
Este detalle no descuenta puntaje en certamen. El profesor lo
notó y aprendió después que su solución inicial no funcionaba siempre.
*/

#include <stdio.h>
#include <pthread.h>

char byte[1];
int done=0, byteFull=0;
pthread_mutex_t dataLock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t byteWritten = PTHREAD_COND_INITIALIZER;
pthread_cond_t byteRead = PTHREAD_COND_INITIALIZER;

void * consumerThread (void * arg) {
   FILE* destFile;
   char local[1];
   destFile = fopen("data.out", "wb");
   while (1) {
      pthread_mutex_lock(&dataLock);
      while (!byteFull)  /* Consumidor espera mientras byteFull es falso */
         pthread_cond_wait(&byteWritten, &dataLock);
      local[0] = byte[0];  /* comsume byte */
      byteFull=0;
      pthread_mutex_unlock(&dataLock);
      pthread_cond_signal(&byteRead);
      if (!done)
         fwrite(local, 1, 1, destFile); /* la parte lenta
         debe estar fuera de la zona crítica para no
         retrasar la otra hebra y perder el efecto buscado */
      else
         break;
```

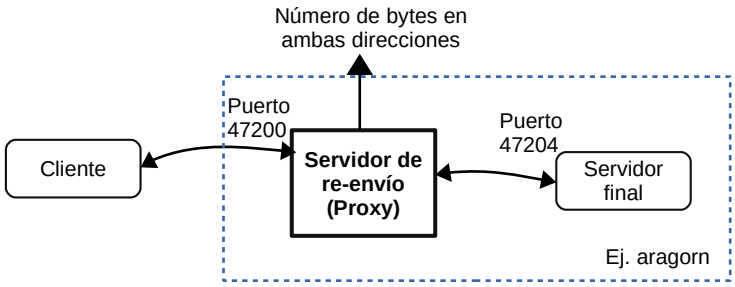
```
    }
    fclose(destFile);
}

int main(int argc, char **argv) {
    int numBytes;
    pthread_t tid;

    pthread_create(&tid, NULL, consumerThread, NULL);
    while(1) {
        pthread_mutex_lock(&dataLock);
        while (byteFull)  /* productor espera mientras byteFull es verdadero */
          pthread_cond_wait(&byteRead, &dataLock);
        numBytes=fread(byte, 1, 1, stdin);
        byteFull=1;
        if (numBytes <=0) {
          done=1;
          pthread_mutex_unlock(&dataLock);
          pthread_cond_signal(&byteWritten);
          break;
        }
        pthread_mutex_unlock(&dataLock);
        pthread_cond_signal(&byteWritten);
    }
    return 0;
}
```

2.- 50 puntos.

Se pide desarrollar un servidor TCP de re-envío (proxy) de datos y que cuente el número de bytes re-enviados en cada dirección. El servidor escucha en puerto 47200. Cuando un cliente se conecta, el servidor establece una conexión hacia el puerto 47204 de la misma máquina. El servidor atiene sólo a un cliente y termina cuando se cierra la conexión por cualquier socket. Para atender a ambos sockets de manera concurrente, el servidor usa el llamado a select. Cada vez que llegan datos por alguno de los sockets, además de re-enviar los datos hacia el otro socket, el servidor muestra por su salida estándar el número de bytes de subida y bytes de bajada re-enviados hasta ese momento.




```
/*
Se pide desarrollar un servidor TCP de re-envío (proxy) de datos y que cuente el número de bytes re-enviados en cada dirección. El servidor escucha en puerto 47200. Cuando un cliente se conecta, el servidor establece una conexión hacia el puerto 47204 de la misma máquina. El servidor atiene sólo a un cliente y termina cuando se cierra la conexión por cualquier socket. Para atender a ambos sockets de manera concurrente, el servidor usa el llamado a select. Cada vez que llegan datos por alguno de los sockets, además de re-enviar los datos hacia el otro socket, el servidor muestra por  su salida estándar el número de bytes de subida y bytes de bajada re-enviados hasta ese momento.
*/
#include <sys/socket.h>  /* socket, bind, listen, accept, connect, recv, send */
#include <arpa/inet.h>   /* htons  */
#include <netinet/in.h>  /* inet_addr */
#include <arpa/inet.h>   /* inet_addr */
#include <string.h>   /* memcpy  */
#include <sys/select.h>  /*  select  */
#include <stdio.h>   /*printf */
#include <unistd.h>   /* close */

int main( int argc, char * argv[]) {
```

```
  int n, len, welcomeSock, fromClientSock, toServerSock;
  struct sockaddr_in server, from, to;
  fd_set readfds, readfdsCopy;
  struct hostent *hp;
  char buf[128];
  int upLoadCounter=0, downLoadCounter=0;

  /* Construct name of socket to send to. */
  server.sin_family = AF_INET;
  server.sin_addr.s_addr = htonl(INADDR_ANY);
  server.sin_port = htons(47200);
  welcomeSock = socket (PF_INET, SOCK_STREAM, 0);
  bind(welcomeSock, (struct sockaddr *)&server, sizeof(server) );
  listen(welcomeSock,4);
  len = sizeof(from);
  printf("Accepting this proxy's client...\n");
  fromClientSock = accept(welcomeSock, (struct sockaddr *)&from, &len);
  close(welcomeSock);

  printf("Cennecting proxy to real server ...\n");
  toServerSock = socket(PF_INET, SOCK_STREAM, 0);
  to.sin_family = AF_INET;
  to.sin_port = htons(47204);
  to.sin_addr.s_addr = inet_addr("127.0.0.1");
  connect(toServerSock, (struct sockaddr *) &to, len);

  FD_ZERO(&readfdsCopy);
  FD_SET(fromClientSock,&readfdsCopy);
  FD_SET(toServerSock,&readfdsCopy);

  for(;;){
    memcpy(&readfds, &readfdsCopy, sizeof(fd_set));
    n = select(FD_SETSIZE, &readfds, (fd_set *) 0, (fd_set *) 0, NULL);
    if (n > 0) {
      if (FD_ISSET(fromClientSock, &readfds)) {
        n=recv(fromClientSock, buf, sizeof(buf), 0);
        if (n <=0 ) break;
        send(toServerSock, buf, n, 0);
        upLoadCounter+=n;
        printf("Accumulated upload traffic: %d [bytes]\n", upLoadCounter);
      }
      if (FD_ISSET(toServerSock, &readfds)) {
        n=recv(toServerSock, buf, sizeof(buf), 0);
        if (n <=0 ) break;
        send(fromClientSock, buf, n, 0);
        downLoadCounter+=n;
        printf("Accumulated download traffic: %d [bytes]\n", downLoadCounter);
      }
    }
  }
  close(fromClientSock);
  close(toServerSock);
}
```