Κωνσταντίνος Σκορδούλης
ΑΜ : 1115 2016 00155

Η εργασία έχει υλοποιηθεί σε C++ (and compiled with g++), βέβαια τα περισσότερα γίνονται με συναρτήσεις/βιβλιοθήκες της C .
Πολύ χρήσιμη ήταν η **snprintf()** για μετατροπή οποιουδήποτε τύπου σε char* και ακριβή δέσμευση μνήμης με length = snprintf(NULL,0, "%typeofdata", data) +1; (για το '/0') .

Source Files: **dropbox_server.cpp, dropbox_client.cpp, Functions.cpp,**

Από **δομές δεδομένων** χρησιμοποίησα **Circular Buffer** και **Single Linked Lists** (4 διαφορετικά είδη λιστών).

Επιπλέον στην εργασία, χρησιμοποίησα κυρίως **Non-Blocking sockets** σε όλη την εργασία (με εξαίρεση στο κομμάτι των **threads** όπου εκεί χρησιμοποίησα **Blocking sockets**).

Σημαντικό1: Σαν **timestamp**, χρησιμοποίησα το **last modification time**, το οποίο είναι διαθέσιμο με την χρήση της **stat()**.

Σημαντικό2: Στην εργασία έγιναν και κάποιες παραδοχές (θα τις αναφέρω σταδιακά).

Παραδοχή1: **IP** → unsigned long int, **port** → unsigned short int

Παραδοχή2: Χρησιμοποίησα **poll()** αντί για **select()** (more on this later).

Signal

Καταρχάς (με τη βοήθεια της **sigaction**) όρισα την συμπεριφορά του dropbox_client και του dropbox_serve, όταν λαμβάνει **SIGINT** (αλλαγή της τιμής ενός flag → 1). Χρησιμοποίησα και το **SA_RESTART**, για να επαναληφθούν system calls τα οποία έχουμε διακόψει.

IP && Listening Port

1) **IP**: Για να βρω την IP διεύθυνση μου, χρησιμοποιώ της εξής συναρτήσεις.
    a. **Gethostname()**, αποθηκεύει σε ένα δικό μας buffer το hostname του υπολογιστή.
    b. **Gethostbyname(),** αποθηκεύει σε μια δομή **struct hostent**, πληροφορίες για την IP μας.
    c. **Inet_ntop(),** μετατρέπει τις πληροφορίες του struct hostent, σε "127.0.1.1" string (το οποίο εκτυπώνω **printf()**).
    d. **Inet_addr()**, διαβάζει το παραπάνω string και το μετατρέπει σε unsigned long int (**NETWORK BYTE ORDER**).

2) **Listening Port**: Συνήθως βάζω σαν port το **0** ➔ αφήνω το λειτουργικό να δεσμεύσει κάποιο **random port**. Για να μάθω όμως πιο port έχει δεσμεύσει τελικά, μετά την **listen()**, χρησιμοποιώ την **getsockname()**, η οποία αποθηκεύει σε ένα **struct sockaddr** , πληροφορίες για το socket (και το port number του➔ **NETWORK BYTE ORDER**).

Lists

Έφτιαξα 4 διαφορετικές λίστες :

1) **List**: η οποία χρησιμοποιείται ως **client list**, δηλαδή ως λίστα ενεργών χρηστών στο δίκτυο μας.
2) **List1**: η οποία χρησιμοποιείται ως **job list** (dropbox_server specific).
   Στον dropbox_server όταν ξεκινάω μια σύνδεση **connect()** ➔ για κάποιο **USER_ON/USER_OFF**, δεν είναι έτοιμη αμέσως.
   Με ενημερώνει (η **poll()**) για το τέλος διαδικασίας με το event **POLLOUT** πάνω στο συγκεκριμένο **file descriptor** .
   Όμως κάπως θα πρέπει να «θυμόμαστε» το μήνυμα, καθώς και το περιεχόμενο του μηνύματος που θα στείλουμε (πληροφορίες για τον εισερχόμενο/εξερχόμενο χρήστη **IP/port**). Αυτή την ανάγκη εξυπηρετεί το job_list.

   ΣΗΜΑΝΤΙΚΟ: Το γεγονός ότι τελείωσε η διαδικασία του connect, δεν σημαίνει απαραίτητα ότι τελείωσε με επιτυχία. Για αυτό ελέγχω με την **getsockopt(** fd, **SO_ERROR,**&variable,sizeof(int)). Αν όλα έχουν πάει σωστά το **variable = 0**, αλλιώς θα επιστρέψει το error number του προβλήματος.

3) **List2**: η οποία χρησιμοποιείται ως **file list** (dropbox_client specific). Κατά την εκκίνηση του client , βάζει όλα τα αρχεία που εμπεριέχονται στο **dirname**, σε αυτή τη λίστα (για να έχουμε γρήγορη πρόσβαση). Το γέμισμα της λίστας γίνεται με τη βοήθεια της **Recursive_Initialize()** (δες Helpful Functions πιο κάτω).

4) **List3**: οποία χρησιμοποιείται ως **pending list** (dropbox_client specific).
   Έστω το scenario, ότι ο **Circular Buffer** είναι γεμάτος και όλα τα threads (main και worker threads) θέλουν να βάλουν κάτι στο Circular buffer ➔ **DEADLOCK**.
   Για αυτό το λόγο, κάθε thread έχει τη δικιά του λίστα (**no need to synchronize**), στην οποία βάζει τις δουλειές που δεν χωρούσαν να μπουν στο buffer (**CNodes*** , τα «κελιά» του buffer ➔ more on this later).

Circular Buffer

Ο circular buffer είναι ένας πίνακας από **CNodes***, δηλαδή πίνακας από δείκτες σε δομή που περιέχει <pathname, timestamp, IP, port>. Στη δομή έχουμε τις εξής πληροφορίες :

1) Int **Size**: Το μέγεθος του πίνακα.
2) Int **Count**: Πόσες θέσεις έχουμε γεμίσει.
3) Int **writeIndex**: Σε ποια θέση του πίνακα, μπορούμε να γράψουμε. Κάθε φορά που εισάγεται καινούργιο στοιχείο, ελέγχουμε αρχικά αν είναι γεμάτος ο Cbuffer . Αν δεν είναι, τότε γράφουμε στη θέση writeIndex και **writeIndex = (writeIndex + 1) % size** (εξου και το κυκλικό size).
4) Int **readIndex**: Σε ποια θέση του πίνακα, μπορούμε να διαβάσουμε. Κάθε φορά που αφαιρούμε στοιχείο, ελέγχουμε αρχικά αν είναι γεμάτος ο Cbuffer. Αν δεν είναι, τότε γράφουμε στη θέση readIndex και **readIndex = (readIndex + 1) % size** (εξου και το κυκλικό size).

Παραδοχή3: Δεν περιορίζω το pathname να είναι 128 bytes. Το έχω αφήσει σαν char*.

Poll()

Καταρχάς φτιάχνουμε έναν **πίνακα** από **struct pollfd**, με αυθαίρετο μέγεθος (50 αν θυμάμαι καλά).
Κάθε φορά που τερματίζουμε ένα socket (success or failure), χρησιμοποιώ την **Poll_update()** (δική μου βοηθητική συνάρτηση) , η οποία κάνει τα εξής:

1) **Array[i].fd = -1** (για να αγνοηθεί από την poll).
2) **Memcpy(array[i],array[fd_counter-1])**, δηλαδή στο «κενό» κελί βάζω την τελευταία εγγραφή (**fd_counter** χρησιμοποιείται για να μην ψάχνω και τις 50 θέσεις ενώ έχω μόνο 5 sockets)
3) **Fd_counter--** (ενημερώνω τον counter μου).

Σε περίπτωση που **γεμίσει ο πίνακας ➔ Poll_Increase_Size()**. Ουσιαστικά κάνω **realloc()** τον πίνακα, με **new_size = size + (size/2)**. Πχ από 50➔75, από 100 ➔ 150.

Τα **events** που κοιτούσα ήταν **POLLIN, POLLOUT, POLLER**. Τα υπόλοιπα όπως POLLHUP κλπ τα χειρίζομαι στο κώδικα (τα οποία βέβαια είναι σχετικά, όχι out of bound data).

1) **POLLER**: Σημαίνει πως κάποιο error συνέβη στο συγκεκριμένο socket (maybe **ETIMEDOUT**). Οπότε κλείνουμε τη σύνδεση.
2) **POLLIN**: Αν μιλάμε για το **listening socket**, τότε υπάρχει κάποια σύνδεση που είναι έτοιμη για **accept()** . Αλλιώς, μας ενημερώνει ότι το συγκεκριμένο socket είναι έτοιμο να διαβαστεί.

   Σημείωση: Το γεγονός ότι είναι έτοιμο να διαβαστεί, δεν σημαίνει ότι υπάρχουν δεδομένα για να διαβάσει, αλλά ότι δεν θα μπλοκάρει η **read()**. Πχ σε non-blocking socket μπορεί να επιστρέψει EAGAIN/EWOULDBLOCK

3) **POLLOUT**: Όταν έχουμε non-blocking **connect()**, επιστρέφει errno = **EINPROGRESS**, δηλαδή ότι η διαδικασία σύνδεσης βρίσκεται σε εξέλιξη. Μόλις τελειώσει η διαδικασία θα ενημερωθούμε με **POLLOUT** .

Σημείωση: Το γεγονός ότι τελείωσε η διαδικασία, δεν μας εξασφαλίζει ότι εκτελέστηκε με επιτυχία. Για αυτό το λόγο χρησιμοποιούμε την **getsockopt(** fd, **SO_ERROR,**&variable,sizeof(int)) η οποία δίνει κάποια τιμή στη μεταβλητή variable.
Αν όλα πήγαν **OK** ➔ **variable = 0**.
Αν προέκυψε **error** ➔ **variable > 0**, που θα αντιστοιχεί σε κάποιο **errno**.

Dropbox_server

Γίνονται λοιπόν τα εξής βήματα:

1) Εκτυπώνουμε την **IP** του μηχανήματος (τρόπος εύρεσης πιο πάνω IP/Listening Port).
2) Φτιάχνουμε το **listening socket**, και έπειτα εκτυπώνουμε και το **Port number** στο οποίο ακούει (τρόπος εύρεσης πιο πάνω IP/Listening Port).
3) Αρχικοποιούμε τις δομές **client_list, job_list,** και τον πίνακα από **struct pollfd.**
4) **While (flag != 1)** (η τιμή του flag αλλάζει με **SIGINT**)
    a. **POLLIN**, διαβάζει το μήνυμα/**string**, με τη βοήθεια της **getString()**. Με λίγα λόγια διαβάζει χαρακτήρα-χαρακτήρα μέχρι να βρει το '**\0**'. Αν τελειώσει ο χώρος του buffer, υπάρξει κάποιο πρόβλημα με τον peer (reached EOF ή ECONNRESET) κλείνουμε τη σύνδεση , ενημερώνουμε τον πίνακα και συνεχίζουμε.
    Ανάλογα με το μήνυμα πράττουμε ανάλογα .
    b. **POLLOUT**, κάποιο socket τερμάτισε τη διαδικασία connection (success or failure).
        i. Ελέγχουμε αν προέκυψε πρόβλημα με την **getsockopt()**.
            1. Αν **όχι**, τότε ψάχνουμε στην **job_list,** για ποιο σκοπό έχει φτιαχτεί το συγκεκριμένο socket **fd**, (**USER_ON**/**USER_OFF**) και στέλνουμε τις πληροφορίες του εισερχόμενου/εξερχόμενου χρήστη.
            2. Αν **ναι**, τότε ψάχνουμε στην **job_list**, σε ποιον προσπαθούσαμε να συνδεθούμε (**dest_IP/dest_port**) και **ξαναπροσπαθούμε**.
5) Τέλος, αφού λάβουμε SIGINT ➔ cleanup and exit.

Παραδοχή4: **LOG_OFF** <IP,port> , δηλαδή ότι στέλνουμε ΚΑΙ την IP/Port (νομίζω επιβεβαιώθηκε στο piazza).

Παραδοχή5: Η απάντηση του GET_CLIENTS ➔ **CLIENT_LIST**, γίνεται πάνω στο ίδιο TCP connection (δεν δημιουργείται καινούργια σύνδεση).

Dropbox_client

Όπως και ο server , δέχεται συνδέσεις από άλλους clients και από τον ίδιο το server με τη χρήση της **poll()**.
Η μόνη διαφορά είναι πως το **main thread** (που διαχειρίζεται την poll()), δεν είναι αρμοδιότητα του να κάνει **connect()** με άλλους clients (για να ανταλλάξει αρχεία).
Με αυτά ασχολούνται τα **Worker threads** (more on this later).
Για αυτό μελετάμε τα events **POLLIN, POLLERR** και μόνο.

Δέχεται 4 είδη μηνυμάτων **GET_FILE_LIST**, **GET_FILE**, **USER_ON**, **USER_OFF**.

Παραδοχή6: Επειδή αναφέρεται στην εκφώνηση ότι, πρέπει να ελέγχω αν ο χρήστης που έστειλε το μήνυμα βρίσκεται στην **client_list**, πρόσθεσα το <IP,port> στην **GET_FILE_LIST** και στην **GET_FILE**.

Παραδοχή7: Για να γνωρίζω πόση ακριβώς μνήμη να δεσμεύσω για το **pathname**, θα στέλνω <length, pathname, timestamp> ,
όπου **length** ➔ unsigned short int (μπορώ να χρησιμοποιήσω την **htons()/ntohs()**) είναι το μήκος του pathname και
timestamp ➔ unsigned long int (**htonl()/ntohl()**).

1. **GET_FILE_LIST <** IP, port>
2. **GET_FILE** < IP, port> < length, path, timestamp>

Worker Threads

Εκτελούν την συνάρτηση **Thread_function()**, και παίρνουν σαν όρισμα τη **struct Args** (δες header file)

Κοινοί πόροι είναι το **client_list**, **Circular_Buffer**. Αυτά τα δύο αποτελούν το **Critical Section** του προγράμματος και για αυτό χρειαζόμαστε **συγχρονισμό**.

Χρησιμοποίησα λοιπόν διάφορες συναρτήσεις για είσοδο και έξοδο από το **CS**:
enterMain(), enterMain1(), enterWorker(), enterWorker1(), exitCS().

Χρησιμοποίησα ένα **pthread_mutex mutex**, **pthread_cond condition**, **bool CS_inside**.
Το mutex για να έχει πρόσβαση στη κοινή μεταβλητή **CS_inside ,** ενώ το condition για να ελέγχω ότι είναι OK οι συνθήκες για να μπει στο CS (I hope you understand).

1. **EnterMain()**: Προσπαθεί να μπει το **main_thread** στο CS, απλά τσεκάρουμε το CS_inside.

2. **EnterMain1():** Χρησιμοποιείται μόνο όταν έχουμε **pending items** (**pending_list->count >0**) και θέλουμε να τα βάλουμε στο Circular Buffer.
   Σε αντίθεση με το παραπάνω :
   a. Δεν θέλουμε να κολλήσουμε στον mutex ➔ **pthread_mutex_trylock()**
   b. Αν περάσουμε, και **CS_inside = true**, πάλι βγαίνουμε (δεν μπλοκάρουμε σε condition variable.
   c. Θέλουμε γενικά να μπούμε κατευθείαν , αλλιώς το παρατάμε.
3. **EnterWorker():** Ελέγχει και αν **CS_inside == true** ΚΑΙ **Cbuffer->empty() ==false .** Χρησιμοποείται όταν θέλει να διαβάσει κάτι από τον buffer
4. **EnterWorker1():** Ελέγχει μόνο **CS_inside == true**, χρησιμοποιείται οπουδήποτε αλλού.

Παραδοχή7: Κάθε thread είναι και **writer** και **reader**, οπότε αποφάσισα μόνο ένας κάθε φορά να μπαίνει (ανεξαρτήτως αν είναι μόνο να διαβάσει πχ από **client_list**).