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: **mirror_client.cpp(main), sender.cpp, receiver.cpp Functions.cpp,**

Από **δομές δεδομένων** χρησιμοποίησα μόνο **Single Linked List** , για να γνωρίζω με ποιους clients συγχρονίστηκα (επιτυχώς) .

Επιπλέον στην εργασία, χρησιμοποίησα **Blocking Named pipes**.

Signals

Καταρχάς (με τη βοήθεια της **sigaction**) όρισα την συμπεριφορά του sender, όταν λαμβάνει **SIGALRM** , **SIGQUIT/SIGINT .** Χρησιμόποιησα και το **SA_RESTART**, για να επαναληφθούν system calls τα οποία έχουμε διακόψει (εκτός αν κάνουμε **exit()** στον handler).

SIGALRM: (Για τα παιδιά-διεργασίες) Για κάθε open/read/write block, αν περάσουν 30 δευτερόλεπτα, τότε ο sender λαμβάνει το σήμα και κάνει τις εξής κινήσεις:

1. Κλείνει το δικιά του μεριά του pipe, και αν είναι ο δημιουργός (**bool creator=true**) τότε κάνει **unlink()** το pipe.
    a. Σε περίπτωση που δεν έχει κλείσει η άλλη πλευρά, δεν υπάρχει πρόβλημα καθώς το pipe γίνεται persist μέχρι να το κλείσει ο άλλος . Γενικά έχουμε 2 περιπτώσεις :
        i. Ο **reader/receiver** να κλείσει το pipe, ενώ ο **writer/sender** το έχει ανοιχτό. Τότε το write() θα χτυπήσει, errno=**EPIPE** (return -1), δηλαδή ότι ο reader έχει κλείσει το pipe , ενώ ο writer περιμένει να γράψει. Παράλληλα, λαμβάνει ο **sender SIGPIPE**, με αποτέλεσμα τον βίαιο τερματισμό του sender. Για αυτό ➔ SIG_IGNORE **SIGPIPE** και το κάνουμε handle στο κώδικα μας (και κάνοντας το σωστό cleanup).
            1. Δηλαδή προέκυψε πρόβλημα κατά τη μεταφορά δεδομένων, **SIGUSR1** στο πατέρα.
        ii. Ο **writer/sender** να κλείσει το pipe, ενώ ο **reader/receiver** το έχει ανοιχτό. Εδώ απλά περιμένουμε να λήξει το όριο των 30 δευτερολέπτων για τον **receiver** ➔ **SIGALRM**, το οποίο γίνεται handle (κάνοντας παράλληλα το cleanup).
2. Στέλνουμε στον πατέρα, **SIGUSR2** , ενημερώνοντας τον ότι το παιδί έλαβε **SIGALRM** και να επαναλάβει την διαδικασία άλλες δύο φορές (σύνολο 3). Δηλαδή να

διακόψει το άλλο παιδί με **SIGINT** και αντίστοιχα να διαγράψει το **mirror/client_ID2** και να επαναλάβει την διαδικασία.

3. Τέλος κάνει ομαλή έξοδος, **exit(1)**.

SIGINT/SIGQUIT: Όταν λάβει αυτό το σήμα γίνεται handle διαφορετικά στα παιδιά και στο πατέρα.

1. Στα παιδιά, γίνεται **cleanup**: closing pipe and **unlink()** (if creator==true).
2. Στο πατέρα, αλλάζουμε την τιμή ενός flag: **quit_flag** = 1 . Αυτό επιτρέπει στον πατέρα να βγει από το main loop (more on this later). Εκτελεί ομαλά το cleanup και return 0.

SIGUSR1: (Αφορά τον πατέρα) Όταν λάβει αυτό το σήμα, τότε καταλαβαίνει ότι κάτι πήγε στραβά στην μεταφορά δεδομένων. Ο handler του αλλάζει μόνο την τιμή ενός flag, **USR1_flag = 1** (έτσι ώστε να μπούμε στο κατάλληλο if) . Επαναλαμβάνει άλλες δύο φορές την διαδικασία, διαγράφοντας το **mirror/client_ID2**.

SIGUSR2: (Αφορά τον πατέρα) Όταν λάβει αυτό το σήμα, τότε καταλαβαίνει ότι ένα από τα παιδιά περίμενε 30 δευτερόλεπτα και δεν πήρε απάντηση. Πάλι ο handler αλλάζει την τιμή ενός flag : **SIGUSR2 = 1**. Επαναλαμβάνει άλλες δύο φορές την διαδικασία, διαγράφοντας το **mirror/client_ID2**.

Sender

Προσπαθεί να δημιουργήσει το pipe (αν υπάρχει απλώς το ανοίγει). Το ανοίγει και μπλοκάρει μέχρι να πάρει απάντηση (δηλαδή να το ανοίξει ο receiver του άλλου client). Στη συνέχεια, εκτελεί την επικοινωνία σύμφωνα με το πρωτόκολλο που μας ζητείται, (με τη βοήθεια της **RecursiveSender()**).

RecursiveSender: Σε αυτή τη συνάρτηση έχουμε ένα αρχικό **char* prefix** (αρχικά prefix= **./input**), το οποίο θα είναι το **relative path** κάθε αρχείου + empty dir . Διασχίζουμε το δέντρο με **DFS.** Γενικότερα προσπαθούμε να χτίσουμε πάνω σε αυτό το prefix όλα τα relative paths { Π.χ ./input/d1/file2, ./input/d1/d2 (d2 is empty) } :

1. Για κάθε dir entry (εκτός από . και ..) φτιάξε το relative path του ως **char* name**. Διακρίνουμε αν είναι αρχείο ή directory με τη βοήθεια της **stat()** και :
    a. Αν είναι **directory**, τότε **recursive call**, με καινούργιο prefix = **char* name**
    b. Αν είναι **file**, τότε ακολουθούμε το πρωτόκολλο :
        i. Μήκος ονόματος/**relative_path** (short int)
        ii. Όνομα (το **name**)
        iii. Μήκος αρχείου, με τη βοήθεια της **stat()**.
        iv. Όλο το αρχείο, διαβάζοντας το αρχείο και γράφοντας στο pipe **byte-byte** (read() / write()).
        v. **Return;**
    c. Παράδειγμα:

i. Για το **αρχείο** ./input/dir1/dir2/file1, το prefix θα αλλάζει στα παρακάτω : ./input ➔ ./input/dir1 ➔ ./ input/dir1/dir2 ➔ ./input/dir1/dir2/file1 (διαδοχικές αναδρομικές κλήσεις)

2. Στην περίπτωση του **empty dir**, δηλαδή να έχουμε να έχουμε «κενό» φάκελο (εκτος από το **.** και το **..**) , κάνουμε σχεδόν το ίδιο :
    i. Μήκος ονόματος/**relative_path** (short int)
    ii. Όνομα (το **prefix!!!**)
    iii. Μήκος αρχείου **= 0** (δικιά μας σύμβαση).
    iv. **Return;**

Σημείωση: Εννοείται πως ό,τι διαβάζουμε/γράφουμε από/για αρχείο, γίνεται **κομματιαστά** με ένα δικό μας **buffer**, του οποίου buffer_size μας δίνεται σαν input.

Αφού τα στείλει όλα, στέλνει **μήκος ονόματος=0** (για να δηλώσει ότι τελείωσε). Τέλος κλείνει το pipe και αν είναι creator το σβήνει.

Receiver

Προσπαθεί να δημιουργήσει το pipe (αν υπάρχει απλώς το ανοίγει). Το ανοίγει και μπλοκάρει μέχρι να πάρει απάντηση (δηλαδή να το ανοίξει ο receiver του άλλου client). Στη συνέχεια, εκτελεί την επικοινωνία σύμφωνα με το πρωτόκολλο που μας ζητείται. Έχουμε ένα **while() loop**, το οποίο διαβάζει το **μήκος αρχείου**, και αν είναι **0 σταματάει**. Έπειτα διαβάζουμε και το όνομα του αρχείου/empty_dir και το επεξεργαζόμαστε με τη βοήθεια της **RecursiveReceiver()** .

RecursiveReceiver: Σε αυτή τη συνάρτηση έχουμε ένα αρχικό **char* construct** (αρχικά prefix= **./mirr/ID2**), το οποίο θα είναι το **relative path** κάθε αρχείου + empty dir . Παράλληλα έχουμε και το **char* remain,** το οποίο περιέχει το υπόλοιπο μονοπάτι (Π.χ **dir1/dir2/file1**). Διασχίζουμε το δέντρο με **DFS.** Γενικότερα προσπαθούμε να χτίσουμε πάνω σε αυτό το construct όλα τα relative paths και εν τέλει να δημιουργήσουμε τα αρχεία +empty directories :

1. Καταρχάς, αφαιρούμε ένα κομμάτι του **remain** με τη χρήση της **strtok()** και το εντάσσουμε στο **construct** μας, αφήνοντας το υπόλοιπο ως έχει (Π.χ **dir2/file1** είναι το καινούργιο **remain**). Έχουμε λοιπόν 2 σενάρια :
    a. Αν το καινούργιο **remain != NULL**, τότε το construct δείχνει προς ένα **Sub-Diretctory**, το οποίο δημιουργούμε με την **mkdir()** με δικαιώματα **777** (read-write-execute, για διάσχιση τουφακέλου). Έπειτα, **αναδρομική κλήση**.
    b. Αν το καινούργιο **remain == NULL**, εκτελούμε την επόμενη read για να μάθουμε το μήκος του αρχείου. Έχουμε 2 σενάρια πάλι :
        i. **Length == 0**, δηλαδή το construct είναι ένας **empty dir** ➔ mkdir() και τελειώσαμε.
        ii. **Length != 0,** δηλαδή το construct είναι **αρχείο** (διαβάζουμε τα περιεχόμενα του αρχείου). Αυτό το επιτυγχάνουμε με ένα while loop ➔ **while(length!=0)** :

1. Γνωρίζουμε το μήκος του αρχείου, οπότε πριν από κάθε **read()** (για το pipe) τσεκάρουμε αν **length < buffer_size**.
    a. Στην περίπτωση αυτή, διαβάζουμε και έπειτα γράφουμε στο αρχείο, length bytes (αντί για buffer_size bytes).
2. Διαβάζουμε από το pipe (Πχ 100 bytes).
3. Έπειτα γράφουμε τα byte αυτά στο αρχείο.
4. Τέλος αφαιρούμε από το length τα bytes που διαβάσαμε ➔ **length = length – bytes**.

2. Παράδειγμα: **dir1/dir2/file3**, **construct** = ./mirror/2, remain= dir1/dir2/file3
    a. Construct: ./mirror/2/dir1, remain= dir2/file3 ➔ mkdir(construct,777)
    b. Construct: ./mirror/2/dir1/dir2, remain= file3 ➔ mkdir(construct,777)
    c. Construct: ./mirror/2/dir1/dir2/file3, remain= **NULL** ➔ Create construct(file)

Αφού τελειώσουμε, κλείνει το pipe και αν είναι creator το σβήνει ➔ **unlink()** .

Mirror_Client(father)

Αφού ορίσουμε τον handler κάθε σήματος, ελέγχουμε τις input παραμέτρους (εννοείται δεν δεχόμαστε λιγότερο από |argc =13| :

1. Τσεκάρουμε καταρχάς ότι το **input, mirror, common** είναι διαφορετικοί φάκελοι, αλλιώς ➔ ERROR.
2. Το **Input**, πρέπει να υπάρχει, else ➔ ERROR.
3. **Mirror, common, log_file**, τα δημιουργούμε.
4. Ελέγχουμε ότι δεν υπάρχει το **ID_file** μέσα στο common, και το δημιουργούμε. Αλλιώς υπάρχει ήδη ο χρήστης στο σύστημα ➔ ERROR.

Στο πρόγραμμα μας χρησιμοποιούμε μια **single linked list,** για να κρατάμε τα **IDs**(char*) αυτών που συγχρονιστήκαμε επιτυχώς.

Από εδώ και πέρα όλα γίνονται σε μια **while(quit_flag != 1)** . Με αυτόν τον τρόπο εξασφαλίζουμε την περιοδική αναζήτηση καινούργιων χρηστών (ή αν έχουν φύγει χρήστες από το σύστημα). Αρχικά **quit_flag = 0** και η τιμή του αλλάζει σε 1 , μόνο αν ο χρήστης δώσει **SIGINT/SIGQUIT**, (έχω βάλει διάφορα if() break; για να βγει εύκολα).

Σε αυτό το loop, κάνει τα εξής:

1. Ελέγχουμε τα **ID_files** μέσα στο **common**, αγνοώντας το **.** (Current folder), **..** (Parent folder), και το δικό μας ID_file. Ελέγχουμε τα υπόλοιπα dir entries για **ID_files** (τελειώνουν με **.id**) , με τη βοήθεια της **strstr()**.Αφού βρούμε κάποιο χρήστη, αρχίζει η διαδικασία του συγχρονισμού (Βήμα 2).
2. Έχουμε μία **for() ,** η οποία 3 φορές προσπαθεί να συγχρονιστεί (αν πετύχει το συγχρονισμό απλά **break**) . **Αποτυχία** θεωρείται :
    a. Κάποιο από τα παιδιά (ή και τα 2) να λάβει **SIGALRM** ➔ πέρασαν 30 sec.
    b. Να πάει κάτι στραβά κατά τη μεταφορά (Πχ ο sender να γράφει και για κάποιο λόγο ο receiver να κλείσει το pipe ➔**SIGPIPE**).

c. Και στις 2 περιπτώσεις διακόπτει το άλλο παιδί και ξαναπροσπαθεί.

3. Μέσα στη for, κάνει τις εξής ενέργειες:
    a. Δημιουργεί { **fork()** } τα 2 παιδιά (**sender** και **receiver**) τα οποία θα εκτελέσουν το δικό τους πρόγραμμα (**exec()**).
    b. Περιμένει (**wait()**) να τελειώσει κάποιο από τα παιδιά και μετά ελέγχει τα 2 flags: **USR1_flag** και **USR2_flag**, για να δει αν προέκυψε κάποιο σφάλμα (και αν ναι ➔ ξαναπροσπαθεί).
    c. Περιμένει για το άλλο παιδί, και αν εξακολουθούν τα flags=0, τότε **bool complete = true;**.
    d. Αφού βγει από το for loop (ελπίζουμε πρόωρα ➔ **success**) ελέγχουμε τη μεταβλητή **bool complete**.
        i. Αν **True**, τότε προσθέτουμε το ID στη λίστα μας , τυπώνουμε το αντίστοιχο μήνυμα και επαναλαμβάνουμε το loop.
        ii. Αν **False**, τότε απλά τυπώνουμε το αντίστοιχο μήνυμα και συνεχίζουμε το loop.
4. Βγαίνοντας από το **while** (κύριο loop) ➔ cleanup memory, ενώ παράλληλα διαγράφουμε **mirror, ID_file**, με τη βοήθεια της **unlink()**.

ΣΗΜΑΝΤΙΚΟ: Πως αντιλαμβανόμαστε ότι ένας χρήστης έχει φύγει?

1. Χρησιμοποιούμε **int counter** (πόσα ID_files υπάρχουν στο φάκελο στο **current loop**) και **List->counter** (με πόσους clients έχουμε συγχρονιστεί).
2. Αν σε κάποιο loop, διακρίνουμε ότι |**counter < List->counter**| , σημαίνει ότι τουλάχιστον ένας χρήστης βγήκε από το σύστημα.
3. Τότε τσεκάρουμε κάθε ID της λίστας αν υπάρχει στο **common**. Αν δεν υπάρχει, το σβήνουμε από τη λίστα και διορθώνουμε τον **List->counter**.