Έχει γίνει το 90% της εργασίας,δηλαδή όλα πέρα του 2ου bash script.
Για το compiling απλά γράφετε "make" στο terminal και make clean
για να κάνετε rm τα εκτελέσιμα,Fifo και txt.
Ας τα πάρουμε με τη σειρά..
JobCommander: Ο jobCommander ξεκινάει,υπολογίζει το μέγεθος των
command line arguments και έπειτα δημιουργεί το 1ο Fifo, αν αυτό
δεν υπάρχει ήδη, όπως και ελέγχει την ύπαρξη του txt αρχείου. Αν
αυτό δεν υπάρχει,αυτό σημαίνει πως ο server δεν είναι
ενεργοποιημένος, συνεπώς κάνουμε exec και fork και ξεκινάει να
εκτελείται και ο server αυτόνομος πλέον.
Έπειτα,δεσμεύομε τα κατάλληλα bytes που θα χρειαστούμε για να
γράψουμε ως απλό κείμενο τα command line arguments. Αμέσως μετά
ο Commander ανοίγει τον Server,στέλνωντας του αρχικά το μέγεθος
του μηνύματος που θα του περάσει και έπειτα το ίδιο το μήνυμα (αυτό
κάνει δυνατή τη δυναμική δέσμευση μνήμης και του buffer που
βρίσκεται στον Server). Κατόπιν, αν το keyword δεν είναι
setConcurrency(το οποίο δεν επιστρέφει τίποτα) ανοίγουμε και το 2ο
φίφο, αυτή τη φορά διαβάζωντας από εκείνο. Πρώτα διαβάζουμε το
μέγεθος του μηνύματος που θα λάβουμε και έπειτα το ίδιο το μήνυμα
(το περιεχόμενο του μηνύματος εξαρτάται από το keyword που
έστειλε ο χρήστης). Εφόσον διαβάζουμε εμείς από το ΦΙΦΟ και θα
είμαστε οι τελευταίοι που θα το χρειαστούμε το κλείνουμε στη
τελευταία γραμμή του commander. Επίσης γίνονται και τα κατάλληλα
free μνήμης (τουλάχιστον στο jobCommander). Να επισημάνω ακόμα
πως όλο αυτό το busy waiting για να βεβαιωθούμε πως ακόμα έχει
τρέξει ο σέρβερ γίνεται επίτηδες,καθώς ο Commander στέλνει σήμαστον σερβερ κάθε
φορά που υπάρχει επικοινωνία μεταξύ τους. (Ο
ακριβής λόγος θα αποκαλυφθεί αργότερα.)
Αυτά τα λίγα από τον jobCommander,πάμε τώρα να δούμε και τον
Server,που είναι και ο κύριος λόγος που θα χρειαστείτε και αυτό το
ReadMe.
Λοιπόν,αρχικά ο σέρβερ αποτελέιται από 2 signal_handlers, 5 global
variables,3(!!!!) διαφορετικές ουρές και μια μεγάαααλη κεντρική λούπα
for() οπού εκεί εκτελούνται και τα features που ζητήθηκαν να
υλοιποιήσουμε. Πάμε πρώτα να δούμε το signal_handler το οποίο
πιάνει sigusr1 signals,που στέλνει ο commander κάθε φορά που
εκτελείται. Ο handler αρχικά στέλνει ένα σήμα SIGCONT ώστε να γίνει
unpause η pause() που υπάρχει στο τέλος της λούπας for(). Το
σημαντικό όμως σε αυτό είναι η αλλαγή της “σημαίας”μεταβλητής
for_read από 0 σε 1. Με αυτό επιτυγχάνεται συγχρονισμένη
επικοινωνία μεταξύ των fifos(Πρώτα fifo από το οποίο λαμβάνει ο
Server το jobCommander μήνυμα “keyword some_word”και έπειτα το
2ο φίφο που στέλνει ο Σερβερ στον Commander, πάλι με όμοιο τρόπο,
πρώτα το μέγεθος έπειτα όλοκληρο το μήνυμα ,εκτός και αν το
keyword που στάλθηκε ήταν το setConcurrency. Για το 2ο σήμα θα
τα πούμε αργότερα. Οπότε προς το παρών έχουμε flag μεταβλητή
for_reading που όταν είναι 1 σημαίνει μεταφορά μηνυμάτων. Όταν
γίνει η πρώτη επικοινωνία με τα fifos, ο σερβερ δέχεται είπαμε το
keyword και όλες τις υπόλοιπες λέξεις μετά. Συνεπώς αν εκτελέσω
./jobCommander issueJob ls ο σερβερ δέχεται το issueJob ls. Στη

συνέχεια σε αυτό το μήνυμα που έχει δεχθεί ο σερβερ το διασχίζει και βρίσκει το πόσα κενά έχει, δηλαδή ξέρει πως το issuejob ls έχει ένα κενό,δηλαδή 2 λέξεις. Έπειτα δημιουργούμε πίνακα από char*(ανάλογα τα spaces που είχε το μήνυμα που δέχθηκε,αυτά+1collumns
θα έχει αυτός ο char* πίνακας,λογικό αφού char[1] έχει 2 collumns,άρα και όταν space=1 τα collumns είναι 2.) Σε αυτό τον πίνακα τοποθετούμε σε κάθε collumn την αντίστοιχη λέξη που είχε το msgbuf που δέχθηκε ο σερβερ από τον κομμαντερ. Πρώτα κάνουμε tokenize το msgbuf αλλά έχουμε φροντίσει να αντιγράφουμε τη κάθε λέξη στο char* data, ώστε να συνεχίσουμε να έχουμε πρόσβαση και αργότερα στα δεδομένα του msgbuf μέσω του data. Το strtok() αλλάζει το string που χωρίζει βάζωντας NULL στο σύμβολο που λειτουργεί ως delimiter για αυτό και κάνουμε τη παραπάνω διαδικασία. Οπότε πλέον τα arguments έχουν σε κάθε collumn μια λέξη από το msgbuf. Συνεπώς το arguments[0] πλέον έχει το keyword(issueJob/poll/stop/exit). Αν το arguments[0] ισούται με το issueJob τότε ορίζουμε πρωτίστως το struct queuejob command, που θα επεριέχει τη τριπλέτα που θα επιστρέψει ο server(+το pid της εκάστοτε διεργασίας) στον κομμαντερ και χρησιμοποιώντας το sprintf δίνουμε το σωστό φορμάτ που θέλουμε για να επιστρέψουμε ενιάιο το μήνυμα μέσα σε μια char* μεταβλητή Serverbuffer. Έπειτα κάνουμε την επικοινωνία μεταξύ του 2ου φίφο που στο οποίο ο σερβερ γράφει το μέγεθος και το ίδιο το μήνυμα και ο κομμάντερ διαβάζει αντίστοιχα το μέγεθος και αμέσως μετά το μήνυμα (είπαμε αυτό το κάνουμε για τη σωστή δέσμευση μνήμης. Σε αυτό το σημείο θυμόμαστε πως το issueJob πέρα της αποστολής της τριπλέτας στον Commander έχει και ως στόχο την εκτέλεση κάποιας διεργασίας. Κάνουμε λοιπόν enqueue στην ουρά που λέγεται queue με τη μεταβλητή data, που θυμόμαστε από πριν εμπεριέχει άθικτο το μήνυμα που παρέλαβε ο σερβερ από τον κομμαντερ. Επειδή είμαστε μερακλήδες όμως (bear with me) κάνουμε και άλλο ενα enqueue αυτή τη φορά στην ουράqueuedprocessesη οποία θα περιέχει τη μεταβλητή
Serverbuffer(θυμόμαστε από πριν πως το Serverbuffer περιείχε formatted τη τριπλέτα <jobID,job,queuePosition> (το γιατί χρειαζόμαστε 2 διαφορετικά enqueue με 2 διαφορετικές μεταβλητές θα επεξηγηθεί αναλυτικά όταν φτάσουμε στο stop feature). Το set_concurrency,το αγαπημένο μου feature, είναι 5 γραμμές κώδικα και το μόνο που κάνει είναι να θέτει τη μεταβλητή concurrent_processes στον αριθμό που βρίσκεται δίπλα από το setConcurrency πχ, “setConcurrency 4”. Η μεταβλητή concurrent_processes χρησιμοποιείται στην διατήρηση της παραλληλίας χωρίς να ξεπερνάει το όριο που έχουμε θέσει. Αν ισούται με 2 τότε θα τρέχουν μαξ 2 διεεργασίες μαζί (αργότερα αυτά).
Στο feature stop, πρέπει να βρούμε το job μιας διεργασίας αρχικά σε ποια κατηγορία ουράς ανήκει, στην ουρά που περιέχει τις ενεργές διεργασίες(αυτή είναι η active_processes ουρά, ή στην ουρά που περιέχει τις διεργασίες που περιμένουν να εκτελεστούν(αυτή είναι η queuejob).
ΣΗΜΑΝΤΙΚΟ: Η χρήση της κάθε ουράς είναι: Ουρά queue, περιέχει σε execvp readable την διεργασία που θέλουμε να εκτελεστεί. Συνεπώς

λίγο πριν εκτελέσουμε τη διεργασία κάνουμε dequeue αυτή την ουρά.
Ουρά queuedprocesses. Περιέχει τις διεργασίες που πρόκειται να εκτελεστούν,πρακτικά τις ίδες με τη queue,μόνο που αυτή τις έχει στη μορφή τριπλέτας. Χρησιμοποιείται στο stop και στο poll που θέλουμε να επιστρέψουμε στον κομμάντερ job_xx και ολόκληρη τη τριπλέτα αντίστοιχα.
active_processes. Περιέχει τις διεργασίες (σε φορματ τριπλέτας) πουτρέχουν αυτή τη στιγμή. Χρησιμοποιείται στα ίδια features με τη queuedprocesses για τους ίδιους λόγος.
Συνεπώς στο stop έχουμε μια flag μεταβλητή. Αν το job που ψάχνουμε βρεθεί στην active_processes ουρά επιστρέφουμε το συγκεκριμένο μήνυμα που χρειάζεται και βγάζουμε το node που περιείχε τη συγκεκριμένη διεργασία.Επίσης,έχωντας φροντίσει όταν εκτελούμε μια διεργασία να κρατάμε το pid της μπορούμε να κάνουμε kill χρησιμοποιώντας το sigterm τερματίζωντας έτσι τη διεργασία.Τέλος θέτουμε και τη flag μεταβλητή να ισούται με 1. Αν δεν βρεθεί και η flag μεταβλητή είναι ακόμα 0 τότε μπαίνουμε στη queuedprocesses,αν τη βρούμε επιστρέφουμε το ανάλογο μήνυμα.
Ένα σχόλιο για αυτό, χρησιμοποιούμε το dequeue_node() που είναι συνάρτηση στο αρχείο dynamicqueue.h. Αυτή η συνάρτηση έχει φτιαχτεί για να μπορεί να κάνει dequeue και nodes της μορφής τριπλέτας (πχ των activeprocesses/queuedcommands ουρών) αλλά και της μορφής job πχ της queue. Έχω βάλει αναλυτικά σχόλια και στον κώδικα για το πότε και τι κάνω dequeue.
Στο poll διασχίζω το κάθε στοιχείο της ουράς αρχικά των active_processes και μετά των queuedprocesses και κάθε φορά περνάω την ουρά που είναι στο φορματ τριπλέτας σε μια μεταβλητή char που την εχω αρχικοποιήσει δυναμικά.
Τέλος το exit στέλνει το απαραίτητο μήνυμα στον commander και κάνει break,κάνοντας έτσι τον server να τερματίσει.Τέλος στην εκτέλεση διεργασιών (που γίνεται ανεξάρτητα αν τα φιφο ετοιμάζονται να επικοινωνήσουν ή οχι) παίρνω την εργασία από το front της queue, τη κάνω tokenize και περνάω τη κάθε λέξη στο char** arguments. Έπειτα κάνω fork,execvp δίνωντας ως ορισματα τον πίνακα arguments και στο πατέρα process φροντίζω να κρατάω το pid και να φτιάχνω κατάλληλα το φορματ για το enqueue των active_processes. Τέλος για την αποφυγή zombie processes,έχω κάνει setup signal_handler για το sigchld το οποίο “πιάνει” όποιο παιδί τερματίζει.
Συνοψίζωντας:
3 ουρές
2 flag variables: for_reading,Queueflag
2 signal_handlers
2 FIFOS
Παραδοχές:
Όταν γίνεται exit συνεχίζει να τρέχει όποια διεργασία βρισκόταν ήδη σε αυτό το στάδιο.
Επίσης το queue υλοποιήθηκε με linked lists,έχει σχετικά σχόλια όπου χρειάζεται.