Περιγραφή Υλοποίησης Εργασίας 1

Ονοματεπώνυμο: Ορφέας Ηλιάδης-Σιβρής

Αριθμός Μητρώου: 1115202000057

Αναφορά Υλοποίησης

multijob.sh		ΝΑΙ	ΜΕΡΙΚΩΣ	ΟΧΙ
allJobsStop.sh		X		
	Εκκίνηση/τερματισμός/ εξαγωγή και έλεγχος ορθότητας εντολη από args	X		
Εκινηση Server	Ελεγχος αν υπαρχει και αν οχι τον ξεκινα	X		
Pipes	Ανοιγμα pipe	X		
	Ενημέρωση Server για αποστολή εντολής	X		
	αποστολή εντολής	X		
	λήψη απάντησης	X		
Γενικά	Ορθή εκκίνηση και δημιουργία jobExecutorServer.txt	X		
	Εκκίνησης ανοιγμα pipe /ληψη μηνύματος για προετοιμασια ληψης /λήψη εντολής / αποστολή απάντησης	X		
	υλοποίηση ουράς /εισαγωγή /εξαγωγή/ διαγραφή στοιχειου/ εκτύπωση όλων	X		
issueJob	Ανάθεση αναγνωριστικού σε εντολή και εισαγωγή στην ουρά	X		
	εκτελεση εντολων με fork	X		
	Διαχείριση SIGCHLD για εκκίνηση επόμενης εντολής από την ουρά	X		
setConcurrency	Έλεγχος εξαγωγή από ουρά κ εντολών ανάλογα με το consurency	X		

	Έλεγχος εκκίνησης νέων εντολών από την ουρά αν μεγάλωσε το consurency	X		
	Έλεγχος μη εκκίνησης νέων εντολών από την ουρά αν μίκρυνε το consurency	X		
stop	Τερματισμός εντολής αν εκτελείται	X		
	Αφαίρεση εντολής από ουρά αν δεν εκτελείται	X		
poll running	Επιστροφή ορθών αποτελεσμάτων	X		
poll queued	Επιστροφή ορθών αποτελεσμάτων	X		
exit	Τερματισμος /αποστολή μηνύματος στο jobCommander / διαγραφή αρχείου	X		

Περιγραφή Κώδικα – Αρχείων

Queue module (Αρχεία queue.h, queue.c) :

Το συγκεκριμένο module χρησιμοποιείται για την υλοποίηση της δομής (waiting_queue) που αποθηκεύει τις δουλειές (jobs) που βρίσκονται σε αναμονή για εκτέλεση.

- Υλοποιήθηκε μια δομή struct Triplet η οποία περιέχει δυο strings ένα για το jobID ένα για το job καθώς και έναν ακέραιο που εκφράζει το position της τριπλέτας στην εκάστοτε δομή (ουρά ή λίστα). Υλοποίηση μια συνάρτησης για αρχικοποίηση δομής triplet και μια καταστροφή της (triplet_create και triplet_destroy αντίστοιχα).
- Υλοποιήθηκε μια δομή κόμβου ουράς (Queue_Node) η οποία περιέχει έναν δείκτη σε μια τριπλέτα και έναν δείκτη στον επόμενο κόμβο της ουράς (next).
- Υλοποιήθηκε μια δομή ουράς Queue η οποία περιέχει έναν front και έναν rear (δείκτη σε) κόμβο. Για ορισμένες βασικές λειτουργίες πάνω στην δομή της ουράς υλοποιήθηκαν οι συναρτήσεις:
  - queue_create : Δεσμεύει μνήμη και αρχικοποιεί μια κενή δομή ουράς (επιστρέφοντας δείκτη σε αυτή) .

- queue_insert_rear: Δέχεται έναν δείκτη σε ουρά και έναν δείκτη σε τριπλέτα και εισάγει την τριπλέτα σην ουρά (προσαρμόζοντας το «position» πεδίο της κατάλληλα)
- queue_remove_front: Δέχεται έναν δείκτη σε ουρά, αφαιρεί από αυτήν την πρώτη τριπλέτα (front) και επιστρέφει δείκτη σε αυτή. Αν η ουρά είναι κενή επιστρέφει NULL. Τα position της κάθε τριπλέτας προσαρμόζονται κατάλληλα.
- queue_remove_jobID: Δέχεται έναν δείκτη σε ουρά, και ένα string (jobID) και βρίσκει και αφαιρεί την τριπλέτα με αυτό το jobID επιστρέφοντας δείκτη σε αυτήν. Αν δεν βρεθεί τριπλέτα επιστρέφει NULL. Τα position της κάθε τριπλέτας προσαρμόζονται κατάλληλα.
- print_queue: Δέχεται μία ουρά από τριπλέτες και την εκτυπώνει.
- queue_destroy: Δέχεται δείκτη σε μία ουρά και την καταστρέφει αποδεσμεύοντας όλη την μνήμη που καταναλώνει.

List Module (Αρχεία list.h , list.c)

Το συγκεκριμένο module χρησιμοποιείται για την υλοποίηση της δομής (running_list) που αποθηκεύει τις δουλειές (jobs) που τρέχουν εκείνη την στιγμή.

- Υλοποιήθηκε μια δομή Match η οποία περιέχει έναν δείκτη σε τριπλέτα και έναν ακέραιο που εκφράζει το process id μιας διεργασίας. Σκοπός της δομής είναι η αντιστοίχιση της τριπλέτας ενός job με το process id της διεργασίας που τρέχει το συγκεκριμένο job. Υλοποιήθηκαν συναρτήσεις match_create και match_destroy για δημιουργία και καταστροφή της δομής.
- Υλοποιήθηκε μια δομή List_Node για τους κόμβους λίστας , η οποία περιέχει έναν δείκτη σε match και έναν δείκτη σε κόμβο λίστας (next).
- Υλοποιήθηκε μια δομή λίστας List οποία περιέχει δύο δείκτες σε κόμβους (first και last) και έναν ακέραιο για το μέγεθος της λίστας. Για τις βασικές λειτουργίες της λίστας υλοποιήθηκαν οι συναρτήσεις:
  - list_create: Δεσμεύει μνήμη και αρχικοποιεί μια κενή δομή λίστας επιστρέφοντας έναν δείκτη σε αυτή.
  - list_append: Δέχεται έναν δείκτη σε λίστα και έναν δείκτη σε match και εισάγει το match στο τέλος της λίστας (προσαρμόζοντας κατάλληλα το «position» πεδίο της τριπλέτας του match)

- list_remove_jobID: Δέχεται έναν δείκτη σε λίστα και ένα string (jobID), βρίσκει και αφαιρεί το match του οποίου η τριπλέτα έχει το jobID που δόθηκε επιστρέφοντας δείκτη σε αυτό. Αν δεν βρεθεί επιστρέφει NULL. Τα position των υπόλοιπων match προσαρμόζονται κατάλληλα.
- list_remove_pid: Δέχεται έναν δείκτη σε λίστα και έναν ακέραιο (pid) , βρίσκει και αφαιρεί το match που έχει το pid που δόθηκε επιστρέφοντας δείκτη σε αυτό. Αν δεν βρεθεί επιστρέφει NULL. Τα position των υπόλοιπων match προσαρμόζονται κατάλληλα.
- list_destroy: Δέχεται έναν δείκτη σε λίστα και την καταστρέφει αποδεσμεύοντας την μνήμη.
- print_list: Δέχεται μία λίστα από match και την εκτυπώνει.

jobCommander :

1) Αρχικά ο jobCommander ελέγχει για την ύπαρξη διεργασίας jobExecutorServer ελέγχοντας αν υπάρχει το text αρχείο jobExecutorServer.txt. Αν αυτό δεν υπάρχει τότε ο jobCommander κάνει fork και στην διεργασία παιδί καλείται μία εντολή exec η οποία «αντικαθιστά» την διεργασία παιδί με τον jobExecutorServer. Αντίθετα αν το text αρχείο υπάρχει (δηλαδή ο jobExecutorServer είναι ήδη ενεργός) τότε ο commander διαβάζει το process id του executor server από αυτό.
2) Στην συνέχεια ο jobCommander αρχικοποιεί δύο pipes (ένα στο οποίο γράφει ο commander και διαβάζει ο server και ένα στο οποίο γράφει ο server και διαβάζει ο commander) χρησιμοποιώντας το process id του στο όνομα τους έτσι ώστε κάθε commander να έχει ένα διαφορετικό ζεύγος από pipes με τα οποία επικοινωνεί με τον server (δηλαδή αν πολλαπλοί commanders επικοινωνούν παράλληλα με τον server να μην χρησιμοποιείται το ίδιο pipe από δυο διαφορετικούς).
3) Έπειτα ο commander στέλνει ένα SIGUSR1 στον server (με kill) το οποίο διαχειρίζεται ο server και έχει σκοπό να τον ενημερώσει για την εισαγωγή μιας εντολής. Αφού σταλθεί, ο commander και ο server ανοίγουν το pipe που γράφει ο commander και ο commander γράφει τα εξής:
   a) Ένα command id που είναι ένα ψηφίο από 1 έως 5 και περιγράφει ποια εντολή στάλθηκε: 1 : issueJob, 2 : setConcurrency, 3 : stop, 4 : poll , 5 : exit.
   b) Τα υπόλοιπα ορίσματα κλήσης του jobCommander χωρισμένα από κενό (whitespace)

c) Στο τέλος τον χαρακτήρα “τέλος κειμένου” (\0) με σκοπό να σταματήσει να διαβάζει ο executor-server.

4) Αφού κλείσουν το pipe ο commander και ο server ανοίγουν το pipe στο οποίο γράφει ο server και μόλις ο server ολοκληρώσει την επεξεργασία γράφει την απάντηση στο pipe, την διαβάζει ο commander και την εκτυπώνει στο output.
5) Αν η εντολή που δόθηκε δεν ήταν έγκυρη ο commander εκτυπώνει σφάλμα χωρίς να επικοινωνήσει με τον server.

jobExecutorServer

1) Ο server ενεργοποιείται όταν κάνει fork και exec ο commander και πρώτα αρχικοποιεί ορισμένες global μεταβλητές (με σκοπό να έχουν άμεση πρόσβαση οι signal handlers) όπως το jobID που περιγράφει τον αύξοντα ακέραιο αριθμό (ξεκινώντας από το 0) που ανατίθεται σε κάθε νέα δουλεία που εισάγεται, το running το οποίο περιγράφει τον αριθμό των διεργασιών που τρέχουν ανά πάσα στιγμή και αρχικοποιείται στο 0 και το concurrency που περιγράφει τον μέγιστο αριθμό ενεργών διεργασιών ανά πάσα στιγμή. Επιπλέον δηλώνονται οι δείκτες στις δομές ουράς και λίστας που θα χρησιμοποιηθούν για τις διεργασίες σε αναμονή και τις ενεργές αντίστοιχα.
2) Στην συνέχεια δημιουργείται το text αρχείο και εισάγεται σε αυτό το process id του server.
3) Αρχικοποιείται η waiting_queue και η running_list και δύο δομές sigaction η μία για διαχείριση του σήματος SIGUSR1 το οποίο στέλνει ο commander όταν υπάρχει νέα εντολή και μία για διαχείριση του σήματος SIGCHLD που στέλνει μια διεργασία παιδί όταν ολοκληρώσει την εκτέλεση της. Στα mask των δύο sigaction εισάγονται όλα τα σήματα ώστε να είναι blocked καθώς εκτελείται κάποιος handler. Ορίζεται ο handler σε κάθε sigaction, ενώ για το sigaction που αφορά την επεξεργασία μηνυμάτων του commander ορίζεται στα flags το SA_SIGINFO που δίνει την δυνατότητα στον handler να διαβάσει το process id της διεργασίας (του εκάστοτε commander) και να το χρησιμοποιήσει για το ορθό άνοιγμα των pipes.
4) Εν συνεχεία εκτελούνται τα 2 sigaction(), δηλαδή ο server μπορεί (πλέον) να διαχειριστεί τα δύο signals (SIGUSR1 και SIGCHLD) , και μπαίνει σε infinite loop περιμένοντας να λάβει κάποιο signal.

commander_signal_handler :

Ο handler εκτελείται όταν ο executor-server λάβει SIGUSR1. Αρχικά αρχικοποιεί τα ονόματα των fifo pipes με βάση το pid του commander που διαβάζει μέσω της παραμέτρου siginfo info. Στην συνέχεια ανοίγει το pipe που γράφει ο commander και διαβάζει από αυτό το πρώτο byte για να προσδιορίσει ποια εντολή δόθηκε (μέσω του command_id).

1) Αν το command id είναι 1 τότε η εντολή είναι issueJob. Διαβάζεται δυναμικά από το pipe η προς εκτέλεση δουλεία (job) και ανατίθεται σε αυτήν ένα καινούργιο id . Αν οι διεργασίες που τρέχουν εκείνη την στιγμή (running) είναι λιγότερες από το concurrency τότε η νέα δουλειά (job) ξεκινάει απευθείας να τρέχει (με fork και exec) και στην συνέχεια δημιουργείται ένα match για την διεργασία και εισάγεται στο running_list. Αντίθετα αν οι διεργασίες που τρέχουν (running) είναι τουλάχιστον όσο το concurrency τότε η διεργασία εισάγεται στην ουρά αναμονής αφού δημιουργηθεί μια τριπλέτα για αυτή. Σε κάθε περίπτωση η εκάστοτε τριπλέτα γράφεται στο pipe που γράφει ο jobExecutorServer για διάβασμα από τον commander.
2) Αν το command id είναι 2 τότε η εντολή είναι setCuncurrency. Διαβάζεται από το pipe η νέα τιμή του concurrency και αν είναι μεγαλύτερη της προηγούμενης, αρχίζουν να αφαιρούνται από την ουρά αναμονής (αν έχει στοιχεία) δουλείες, να εκτελούνται (με fork και exec) και να εισάγονται τα αντίστοιχα match στην running_list. Τέλος το concurrency αλλάζει και παίρνει την νέα τιμή.
3) Αν το command id είναι 3 τότε η εντολή είναι stop. Διαβάζει από το pipe το jobID προς αφαίρεση. Το αναζητά αρχικά στην waiting_queue και αν το βρει το αφαιρεί. Αν δεν το βρει το ψάχνει στην running list και αν το βρει κάνει kill την αντίστοιχη διεργασία με SIGKILL (Η αφαίρεση από την λίστα και η εκτέλεση νέας δουλειάς που ήταν σε αναμονή γίνονται από τον child signal handler όταν σταλθεί το SIGCHLD από την διεργασία που τελειώνει). Σε κάθε περίπτωση γράφει την τριπλέτα που αφαιρέθηκε στο pipe που γράφει ο server ή αν δεν βρέθηκε ούτε στην λίστα ούτε στην ουρά γράφει αντίστοιχο μήνυμα.
4) Αν το command id είναι 4 τότε η εντολή είναι poll. Διαβάζει από το pipe το όρισμα που δόθηκε. Αν το όρισμα είναι “queued” προσπελαύνει την ουρά

αναμονής γράφοντας όλες τις τριπλέτες στο pipe. Αν το όρισμα είναι “running” προσπελαύνει την λίστα με τις διεργασίες που τρέχουν και γράφει στο pipe τις τριπλέτες από κάθε στοιχείο της (match). Αν το όρισμα είναι οτιδήποτε άλλο γράφει μήνυμα λάθους.

5) Αν το command id είναι 5 τότε η εντολή είναι exit. Ο server γράφει ένα μήνυμα στο pipe για να ενημερώσει τον commander για την ολοκλήρωση του. Καταστρέφει τις δομές ουράς και λίστας (waiting_queue, running_list) και απελευθερώνει την μνήμη και διαγράφει το text αρχείο (jobExecutorServer.txt). Τέλος εκτελεί την exit() και ο server ολοκληρώνει την εκτέλεση του.

child_signal_handler:

Ο handler αυτός εκτελείται όταν σταλθεί στον server SIGCHLD, δηλαδή όταν μια διεργασία ολοκληρώσει την εκτέλεση της ή όταν γίνει «stopped» από τον server. Αρχικά βρίσκει τον κόμβο της λίστας που αφορά την διεργασία που έστειλε το signal (με χρήση waitpid και WNOHANG flag και προσπέλαση της λίστας). Εν συνεχεία αφαιρεί το αντίστοιχο match από την λίστα και αν οι διεργασίες που τρέχουν εκείνη την στιγμή (running) είναι λιγότερες από το concurrency τότε αρχίζουν να αφαιρούνται από την ουρά αναμονής και να εκτελούνται όσες δουλείες επιτρέπει το concurrency.

Παραδοχές:

- Η δομή που χρησιμοποιείται για να αποθηκεύει τις δουλειές (jobs) που τρέχουν είναι λίστα ενώ η δομή που αποθηκεύει τις δουλειές σε αναμονή για εκτέλεση είναι ουρά.
- Τα jobID ξεκινούν από το job_0, κάθε αριθμός δίνεται σε μοναδική δουλειά και κάθε δουλεία παίρνει για id τον αμέσως μεγαλύτερο αριθμό από την τελευταία δουλειά που δόθηκε στο σύστημα.
- Τα fifo pipes συμπεριλαμβάνουν στο όνομα τους το pid του εκάστοτε jobCommander ώστε κάθε φορά να δημιουργείται και να χρησιμοποιείται ένα unique ζεύγος από pipes (ένα για write και ένα για read) για επικοινωνία με τον

server. Αυτό έχει σκοπό να μην δημιουργηθεί πρόβλημα σε περίπτωση που πολλαπλοί commanders προσπαθούν να επικοινωνήσουν με τον server μέσω των pipes ταυτόχρονα.

- Το concurrency αρχικοποιείται σε 3, πράγμα που σημαίνει ότι by default μπορούν να τρέχουν 3 διεργασίες ταυτόχρονα.
- Στην τριπλέτα που επιστρέφει η εντολή issueJob (ή και η poll) το 3ο στοιχείο είναι το position ΕΙΤΕ στην ουρά (αν η δουλειά μπήκε στην αναμονή) ΕΙΤΕ στην λίστα (αν η δουλειά εκτελέστηκε κατευθείαν).
- Ο jobCommander εκτελείται με χρήση “./”. Δηλαδή λ.χ. : ./jobCommander poll running.

Bash scripts:

multijob.sh: Προσπελαύνει τα ορίσματα κλήσης και για κάθε ένα: αν υπάρχει (εφόσον πρόκειται για αρχεία) και αν έχουμε read rights διαβάζονται μια-μια οι εντολές (μια ανά γραμμή) και εκτελούνται με χρήση jobCommander issueJob.

allJobsStop.sh: Εκτελεί poll running και poll queued και αποθηκεύει τα αποτελέσματα σε 2 μεταβλητές. Στην συνέχεια παίρνει την πρώτη στήλη από την κάθε μεταβλητή (την στήλη που περιέχει τα jobIDs) και για κάθε jobID εκτελεί jobCommander stop.

Έλεγχος ορθότητας-Testing

- Αρχεία test_jobExecutor_* : Τα ενδεικτικά test που δόθηκαν, ελαφρώς τροποποιημένα στις προδιαγραφές της υλοποίησης (μαζί και το αρχείο progDelay.c).
- Εντολή ./multijob.sh commands_1.txt commands_2.txt και Αρχείο bash_script_test_2.sh: Τα ενδεικτικά test που δόθηκαν για τα bash scripts (ελαφρώς τροποποιημένα στις προδιαγραφές της υλοποίησης).
- Επιπλέον ένα αρχείο sleep_test_1.c και 2 text αρχεία: multijob_test_1.txt multijob_test_2.txt τα οποία χρησιμοποιήθηκαν σε επιπλέον ελέγχους.