Σκοπός

Στόχος αυτής της εργασίας είναι η εξοικείωσή σας με τις κλήσεις συστήματος της οικογένειας exec*(), της fork(), όπως επίσης και με κλήσεις συστήματος χαμηλού επιπέδου για τη διαχείριση αρχείων και το bash scripting.

Θα υλοποιήσετε μία εφαρμογή που έχει ως σκοπό την εκτέλεση εργασιών που λαμβάνει από κάποιο αρχείο εισόδου. Η εφαρμογή αυτή θα προσφέρει στο χρήστη πληροφορία αλλά και τη δυνατότητα να διαχειριστεί το σύνολο των υπό εκτέλεση διεργασιών. Στη συνέχεια, θα γράψετε μία σειρά από bash scripts τα οποία θα προσφέρουν στο χρήστη πιο σύνθετες δυνατότητες.

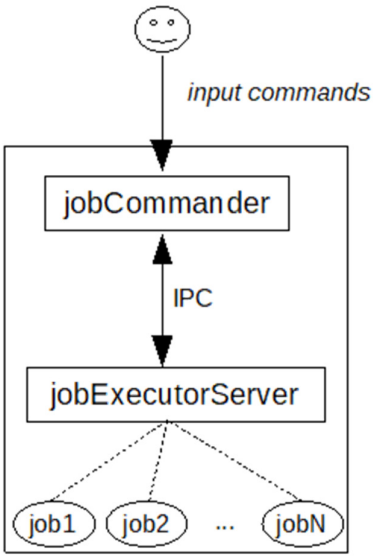
Εφαρμογή jobExecutor (βάρος: 80%)

Αναπτύξτε την εφαρμογή jobExecutor η οποία θα αναλαμβάνει την εκτέλεση εργασιών (jobs) που θα δίνονται από το χρήστη. Η δουλειά του jobExecutor είναι να βάζει εργασίες (jobs) να τρέξουν, αλλά με έλεγχο του ρυθμού ροής τους (flow control). Για το λόγο αυτό, βασική έννοια είναι ο τρέχων βαθμός παραλληλίας (concurrency). Για παράδειγμα, ας θεωρήσουμε ότι ο βαθμός παραλληλίας είναι 4. Αυτό σημαίνει ότι οι 4 πρώτες διαθέσιμες εργασίες που έχουν υποβληθεί στην εφαρμογή (ή όσες υπάρχουν, αν έχουν υποβληθεί λιγότερες από 4) θα πρέπει να εκτελούνται, με τις υπόλοιπες σε ουρά αναμονής. Οι εργασίες σε ουρά αναμονής ΔΕΝ έχουν process id στο λειτουργικό σύστημα: ο jobExecutor πολύ απλά δεν έχει ξεκινήσει να τις εκτελεί ακόμα. Αν μία εργασία από τις 4 τερματίσει, τότε ο jobExecutor θα αρχίσει να εκτελεί την πρώτη από την ουρά αναμονής. Αν ο βαθμός παραλληλίας αλλάξει, ο jobExecutor θα προσαρμοστεί αντίστοιχα, χωρίς να σταματήσει εργασίες που ήδη τρέχουν, σε περίπτωση μείωσης του βαθμού παραλληλίας.

Η εφαρμογή (jobExecutor) θα προσφέρει τη λειτουργικότητά της στο χρήστη μέσω δύο διεργασιών,

- του jobCommander μέσω του οποίου ο χρήστης αλληλεπιδρά με την εφαρμογή και
- του jobExecutorServer που αναλαμβάνει την διαχείριση και εκτέλεση των εντολών

Στο Σχήμα 1 παρουσιάζεται μία αναπαράσταση της δομής της εφαρμογής.



Σχήμα 1: Σχηματική αναπαράσταση της αρχιτεκτονικής του jobExecutor.

jobCommander

Ο jobCommander δίνει τη δυνατότητα στον χρήστη να αλληλεπιδράσει με την εφαρμογή μέσω απλών εντολών. Οι εντολές δίνονται ως ορίσματα κατά την κλήση του jobCommander.

Οι εντολές του jobCommander είναι οι ακόλουθες. Παρακαλείσθε για την χρήση των εντολών όπως ακριβώς προδιαγράφονται ώστε να περνούν από πιθανά scripts που θα χρησιμοποιηθούν κατά την εξέταση. Δεν θα βαθμολογηθούν εντολές με διαφορετικές προδιαγραφές.

1. **issueJob <job>:** Μέσω αυτής της εντολής εισάγονται εργασίες στο σύστημα που πρόκειται να εκτελεστούν. Το job είναι μια συνηθισμένη γραμμή εντολών Unix. Ο jobExecutorServer αναθέτει μια μοναδική αναγνωριστική τριπλέτα (<jobID,job,queuePosition>) στην εργασία, Το queuePosition αντιστοιχεί στη αύξουσα θέση (integer χωρίς leading zeros) που τοποθετήθηκε η εργασία στην ουρά. Το JobID θα πρέπει να ακολουθεί το πρότυπο **job_XX**. όπου XX ένας αύξων μοναδικός αριθμός (χωρίς leading zeros) που αυξάνεται για κάθε νέα εργασία που δέχεται ο jobExecutorServer. Η τριπλέτα <jobID,job,queuePosition> επιστρέφεται στον jobCommander ό οποίος την εκτυπώνει στην κονσόλα. Η ουρά δεν έχει περιορισμό μεγέθους.
2. **setConcurrency <N>:** Η παράμετρος αυτή θέτει το βαθμό παραλληλίας, δηλαδή το μέγιστο αριθμό ενεργών εργασιών που μπορεί να εκτελεί ανά πάσα χρονική στιγμή η εφαρμογή (δεδομένου ότι υπάρχουν διαθέσιμες). Η προκαθορισμένη τιμή είναι 1. Η εντολή μπορεί να αποσταλεί και κατά τη διάρκεια εκτέλεσης εργασιών και θα αλλάξει την συμπεριφορά του εξυπηρέτη από την λήψη της και μετά. Δεν χρειάζεται να επιστραφεί κάτι στον jobCommander.
3. **stop <jobID>:** Τερματίζεται η εκτέλεση της εργασίας με το συγκεκριμένο αναγνωριστικό (job_XX). Σε περίπτωση που δεν εκτελείται, αλλά είναι υποψήφια για μελλοντική εκτέλεση, αφαιρείται από την ουρά των υπό εκτέλεση εργασιών. Επιστρέφεται μήνυμα στον jobCommander ως εξής
   job_XX removed σε περίπτωση που ήταν στην ουρά
   job_XX terminated σε περίπτωση που εκτελούνταν.

Ο jobCommander εκτυπώνει τα αντίστοιχα μηνύματα στην κονσόλα.

4. **poll [running,queued]:**
    - poll running : Για κάθε εργασία που είναι υπό εκτέλεση (running) αυτή τη χρονική στιγμή επιστρέφει την τριπλέτα <jobID,job,queuePosition> την οποία ο jobCommander εκτυπώνει στην κονσόλα.
    - poll queued : για κάθε εργασία που είναι σε κατάσταση αναμονής (queued), επιστρέφει την τριπλέτα <jobID,job,queuePosition> την οποία ο jobCommander εκτυπώνει στην κονσόλα..
5. **exit:** Τερματίζεται η λειτουργία του jobExecutorServer. Ο jobExecutorServer πριν τερματίσει επιστρέφει στον jobCommander το μήνυμα
   jobExecutorServer terminated.
   το οποίο και εκτυπώνεται στην κονσόλα.

Ακολουθούν παραδείγματα των ανωτέρω εντολών με την λειτουργικότητα που πρέπει να υποστηρίζετε στην υλοποίηση σας.

issueJob

```
jobCommander issueJob ls -l /path/to/directory1
jobCommander issueJob wget aUrl
jobCommander issueJob grep "keyword" /path/to/file1
jobCommander issueJob cat /path/to/file2
jobCommander issueJob ./executable arg1 arg2
```

setConcurrency

```
jobCommander setConcurrency 4

jobCommander setConcurrency 1
```

stop

```
jobCommander stop job_1

jobCommander stop job_15
```

poll

```
jobCommander poll running

jobCommander poll queued
```

exit

```
jobCommander exit
```

και ένα σενάριο εκτέλεσης μπορεί να είναι:

jobCommander issueJob ls -l /home/users/nikosp/SysPro2024Data

jobCommander issueJob wget http://rhodes.mm.di.uoa.gr:8888/others/genome/chr19.fa

jobCommander issueJob grep "AATGGG" /home/users/nikosp/SysPro2024Data/gene.fasta

jobCommander issueJob cat /home/users/nikosp/SysPro2024Data/gene.fasta

jobCommander issueJob ./progDelay 5

όπου progDelay είναι το εκτελέσιμο για παράδειγμα του παρακάτω

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <number>\n", argv[0]);
        return 1;
    }
    int delay = atoi(argv[1]);
    if (delay <= 0) {
        printf("Invalid number: %s\n", argv[1]);
        return 1;
    }
    for (int i = 0; i < delay; i++) {
        sleep(1);
        printf("$");
        fflush(stdout);
    }
    printf("\n");
    return 0;
}
```

Ο κύκλος ζωής του jobCommander είναι μία μόνο εντολή. Δηλαδή, η εκτέλεσή του ολοκληρώνεται όταν μεταβιβάσει με επιτυχία την εντολή στον jobExecutorServer και λάβει το μήνυμα απάντησης από τον jobExecutorServer όπου απαιτείται.

Μέθοδος Υλοποίησης

Κάθε φορά που εκτελείται ένας jobCommander πρέπει να είναι σε θέση να αναγνωρίσει αν ο jobExecutorServer είναι ενεργός ή όχι, εξετάζοντας την ύπαρξη ενός αρχείου jobExecutorServer.txt (). Το αρχείο αυτό το δημιουργεί ο jobExecutorServer και περιέχει το αναγνωριστικό της διεργασίας (process id/pid) του server. Σε περίπτωση που δεν είναι ενεργός ο jobExecutorServer, θα πρέπει να τον δημιουργήσει ο jobCommander. Ο jobExecutorServer θα πρέπει να διαγράφει το αρχείο jobExecutorServer.txt κατά τον τερματισμό του.

Η επικοινωνία ανάμεσα στις δύο διεργασίες λαμβάνει χώρα μέσω named pipes (κλήση συστήματος mkfifo()). Η προκαθορισμένη συμπεριφορά των named pipes είναι να μπαίνει σε κατάσταση αναμονής η διεργασία που ανοίγει το ένα άκρο μέχρι να ανοιχτεί η σωλήνωση και από το άλλο άκρο. Βέβαια, μπορούμε να αποφύγουμε την παραπάνω συμπεριφορά αν θέσουμε το O_NONBLOCK flag στο δεύτερο όρισμα της κλήσεως συστήματος open(). Για παράδειγμα, αν θέλουμε να ανοίξουμε ένα named pipe για ανάγνωση χωρίς να τεθούμε σε αναμονή, κάνουμε την κλήση open(pipe_name , O_RDONLY | O_NONBLOCK). Προσοχή, αν σε αυτήν την περίπτωση ανοιχτεί το named pipe για γράψιμο (από το ένα άκρο) και δεν είναι ανοιχτό για διάβασμα (από το άλλο άκρο), τότε θα επιστραφεί σφάλμα. Είστε ελεύθεροι να διαλέξετε όποια μέθοδο λειτουργίας των σωληνώσεων θέλετε. Θα πρέπει να υπάρχουν named pipes για γράψιμο και από τις δύο πλευρές.

Ο jobCommander πρέπει με κάποιο τρόπο να ενημερώνει τον jobExecutorServer ότι σκοπεύει να του μεταβιβάσει κάποια εντολή μέσω της σωλήνωσης έτσι ώστε ο δεύτερος να πάει να τη διαβάσει. Αυτό, για παράδειγμα, θα μπορούσε να επιτευχθεί με την αποστολή κάποιου προσυμφωνημένου σήματος (signal-software interrupt).

Ο jobExecutorServer αρχικά είναι αδρανής μέχρι να δεχτεί κάποια/ες εντολή/ες (μέσω issueJob), όπου ξεκινά την εκτέλεσή τους, λαμβάνοντας υπόψη τον βαθμό παραλληλίας.

Για κάθε εργασία που λαμβάνει προς εκτέλεση,, της αναθέτει ένα μοναδικό αναγνωριστικό, όπως προδιαγράφηκε σε προηγούμενη παράγραφο και την τοποθετεί στην ουρά στην θέση queuePosition ανάλογα με την πληρότητα που έχει εκείνη την ώρα. Στον jobCommander επιστρέφεται η τριπλέτα <jobID,job,queuePosition>.

Όταν η εργασία πρέπει να εκτελεστεί, αφαιρείται από την ουρά, δημιουργείται μία διεργασία-παιδί μέσω της κλήσεως συστήματος fork() η οποία αμέσως εκτελεί την εργασία μέσω κάποιας εκ των συναρτήσεων της οικογένειας exec*() (execv(),execvp κλπ.).

Κατά την ολοκλήρωση μιας διεργασίας-παιδί, ο πυρήνας του Linux αποστέλλει στην γονική της διεργασία το σήμα (signal) SIGCHLD. Ο server σας θα πρέπει να διαχειρίζεται αυτό το σήμα έτσι ώστε να ενημερωθεί για τον τερματισμό κάποιας διεργασίας-παιδί, Όταν λάβει το SIGCHLD, θα πρέπει να πάρει από την ουρά αναμονής την επόμενη εργασία και να τη θέσει υπό εκτέλεση (δεδομένου ότι υπάρχει εργασία διαθέσιμη).

Συνολικά, ο jobExecutorServer πρέπει να επικοινωνεί με τον jobCommander μέσω named pipe καθώς και να ενημερώνεται για τον τερματισμό των διεργασιών-παιδιών του μέσω του σήματος (signal) SIGCHLD.

Bash Scripting (βάρος: 20%)

Αναπτύξτε τα ακόλουθα bash scripts, τα οποία έχουν ως σκοπό να χρησιμοποιήσουν την απλή διεπαφή του jobExecutor με σκοπό να παρέχουν στο χρήστη λειτουργίες υψηλότερου επιπέδου:

1. multijob.sh <file1> <file2> ... <fileN>: Το script αυτό λαμβάνει σαν είσοδο αρχεία τα οποία περιέχουν εργασίες προς εκτέλεση, μία ανά γραμμή. Σκοπός είναι να εισαχθούν όλες αυτές οι εργασίες στο σύστημα.

2. allJobsStop.sh: Το script αυτό σταματά όλες τις εργασίες που υπάρχουν ενεργές στον jobExecutor, (είτε είναι υπό εκτέλεση είτε στην ουρά αναμονής ,).

Διαδικαστικά

● Το πρόγραμμά σας θα πρέπει να γραφεί σε C (ή C++) και σας θα πρέπει να τρέχει στα Linux workstations του Τμήματος. Κώδικας που δε μεταγλωττίζεται εκεί, θεωρείται ότι δεν μεταγλωττίζεται. Δε θα γίνει αποδεκτή η εξέταση της εργασίας σε άλλον υπολογιστή.

● Για επιπρόσθετες ανακοινώσεις, παρακολουθείτε το forum του μαθήματος στο piazza.com. Η πλήρης διεύθυνση είναι https://piazza.com/uoa.gr/spring2024/k24/home. Η παρακολούθηση του φόρουμ στο Piazza είναι υποχρεωτική.

● Ο κώδικάς σας θα πρέπει να αποτελείται από τουλάχιστον δύο (και κατά προτίμηση περισσότερα) διαφορετικά αρχεία. Η χρήση του separate compilation είναι επιτακτική και ο κώδικάς σας θα πρέπει να έχει ένα Makefile.

● Βεβαιωθείτε πως ακολουθείτε καλές πρακτικές software engineering κατά την υλοποίηση της άσκησης. Η οργάνωση, η αναγνωσιμότητα και η ύπαρξη σχολίων στον κώδικα αποτελούν κομμάτι της βαθμολογίας σας.

● Η υποβολή θα γίνει μέσω eclass.

● Ο κώδικάς σας θα πρέπει να κάνει compile στα εκτελέσιμα jobCommander και jobExecutorServer όπως **ακριβώς** ορίζει η άσκηση.

Τι πρέπει να παραδοθεί

● Όλη η δουλειά σας (πηγαίος κώδικας, Makefile και README) σε ένα tar.gz file με ονομασία OnomaEponymoProject1.tar.gz. Προσοχή να υποβάλετε μόνο κώδικα, Makefile, README και όχι τα binaries. Η άσκησή σας θα γίνει compile από την αρχή πριν βαθμολογηθεί.

● Όποιες σχεδιαστικές επιλογές κάνετε, θα πρέπει να τις περιγράψετε σε ένα README (απλό text αρχείο) που θα υποβάλετε μαζί με τον κώδικά σας. Το README χρειάζεται να περιέχει μια σύντομη και περιεκτική εξήγηση για τις επιλογές που έχετε κάνει στον σχεδιασμό του προγράμματός σας σε 1-2 σελίδες ASCII κειμένου. Συμπεριλάβετε την εξήγηση και τις οδηγίες για το compilation και την εκτέλεση του προγράμματός σας.

● Ο κώδικας που θα υποβάλετε θα πρέπει να είναι δικός σας. Απαγορεύεται η χρήση κώδικα που δεν έχει γραφεί από εσάς ή κώδικας που έχει γραφτεί με τη βοήθεια μηχανών τύπου chatGPT.

● Καλό θα είναι να έχετε ένα backup .tar της άσκησής σας όπως ακριβώς αυτή υποβλήθηκε σε κάποιο εύκολα προσπελάσιμο μηχάνημα (server του τμήματος, github, cloud).

● Η σωστή υποβολή ενός σωστού tar.gz που περιέχει τον κώδικα της άσκησής σας και ό,τι αρχεία χρειάζονται είναι αποκλειστικά ευθύνη σας. **Άδεια tar/tar.gz ή tar/tar.gz που έχουν λάθος και δε γίνονται extract δε βαθμολογούνται**.

Τι θα βαθμολογηθεί

● Η συμμόρφωση του κώδικά σας με τις προδιαγραφές της άσκησης.
● Η οργάνωση και η αναγνωσιμότητα (μαζί με την ύπαρξη σχολίων) του κώδικα.
● Η χρήση Makefile και το separate compilation.

Άλλες σημαντικές παρατηρήσεις

● Οι εργασίες είναι ατομικές.
● Όποιος υποβάλλει / δείχνει κώδικα που δεν έχει γραφτεί από την ίδια/τον ίδιο **μηδενίζεται** στο μάθημα.
● Αν και αναμένεται να συζητήσετε με φίλους και συνεργάτες το πώς θα επιχειρήσετε να δώσετε λύση στο πρόβλημα, αντιγραφή κώδικα (οποιασδήποτε μορφής) είναι κάτι που δεν επιτρέπεται. Οποιοσδήποτε βρεθεί αναμεμειγμένος σε αντιγραφή κώδικα απλά παίρνει μηδέν στο μάθημα. Αυτό ισχύει για όσους εμπλέκονται ανεξάρτητα από το ποιος έδωσε/πήρε κλπ. Τονίζουμε πως θα πρέπει να λάβετε τα κατάλληλα μέτρα ώστε να είναι προστατευμένος ο κώδικάς σας και να μην αποθηκεύεται κάπου που να έχει πρόσβαση άλλος χρήστης (π.χ., η δικαιολογία «Το είχα βάλει σε ένα github repo και μάλλον μου το πήρε από εκεί», δεν είναι δεκτή.)
● Οι ασκήσεις προγραμματισμού μπορούν να δοθούν με καθυστέρηση το πολύ 3 ημερών και με ποινή 5% για κάθε μέρα αργοπορίας. Πέραν των 3 αυτών ημερών, δεν μπορούν να κατατεθούν ασκήσεις.
● Το πρόγραμμά σας θα πρέπει να γραφτεί σε C ή C++. Μπορείτε να χρησιμοποιήσετε μόνο εντολές οι οποίες είναι διαθέσιμες στα μηχανήματα Linux του τμήματος. Πρόγραμμα που πιθανόν μεταγλωτίζεται ή εκτελείται στο προσωπικό σας υπολογιστή αλλά όχι στα μηχανήματα Linux του τμήματος, θεωρείται ότι δε μεταγλωτίζεται ή εκτελείται αντίστοιχα.