

2ο Project 2023-2024, Αναφορά Ομάδας

Παντελής Φλουρής, up1093507, up1093507@ac.upatras.gr
Αγγελος Μενεγάτος, up1093426, up1093426@ac.upatras.gr
Χρυσάφης Κολτσάκης, up1084671, up1084671@ac.upatras.gr
Γιώργος Αμαξοπουλος, up1093311, up1093311@ac.upatras.gr

Λειτουργούν όλα τα υποερωτήματα και στα 2 μέρη.

Βοηθητικές συναρτήσεις – queue (functions.c)

- Υλοποιήσαμε μια ουρά, η οποία κρατάει στα nodes της το PID της διεργασίας που είναι συνδεδεμένη μαζί της (κάθε node αντιστοιχεί σε μια εντολή που εκτελείται από μια διεργασία-παιδί).
- Χρησιμοποιήσαμε την συνάρτηση `get_wtime(void)` (που μας ειχατε δώσει στο πρώτο project) για χρονικούς υπολογισμούς.
- Για λόγους ευκολίας, δημιουργήσαμε μια συνάρτηση που ελέγχει εάν η ουρά μας είναι κενή.

Τα nodes περιέχουν πληροφορίες για το PID της διεργασίας, την εντολή που εκτελούν, την κατάσταση της διεργασίας και έναν δείκτη στο επομένο node.

Πρώτη φάση – scheduler.c

Αφού αρχικοποιήσουμε τα απαραίτητα (συμπεριλαμβανομένων του κενού handler για το σήμα `SIGCHLD` και της παγκόσμιας μεταβλητής `start_time`), το πρώτο μας μέλημα είναι να διαβάσουμε την είσοδο του χρήστη περί είδος δρομολόγησης, `time_slice` και αρχείου που περιέχει τις εφαρμογές. Αυτό γίνεται εύκολο με αξιοποίηση του `argv[]`.

Επειτα διαβάζουμε γραμμή γραμμή τις εντολές του αρχείου που μας είχε υποδείξει ο χρήστης, και κάνουμε `enqueue` με placeholder τιμή `PID = 0` (η συσχέτιση του Node με διεργασία θα γίνει στην συνάρτηση `χρονοπρογραμματισμού`), και αρχικό `status NEW`.

Στην συνέχεια, σε περίπτωση που ο χρήστης επέλεξε `FCFS`, καλείται η συνάρτηση `RR` με `time slice = 1000s` (επεξήγηση για αυτή την επιλογή παρακάτω), και εάν ο χρήστης επέλεξε `RR`, καλείται η συνάρτηση `RR` με το `time_slice` που ορίστηκε από τον χρήστη.

Η συναρτηση RR λειτουργει ως εξης:

1. Δημιουργει ενα queue στο οποιο θα αποθηκευονται οι εντολες που εχουν ολοκληρωθει
2. Μπαινει σε εναν βρογχο επαναληψης, ο οποιος επαναλαμβανεται εως οτου η αρχικη ουρα να ειναι αδεια.
3. Αφαιρει το πρωτο στοιχειο της ουρας και το αποθηκευει σε μια μεταβλητη `current_node`
4. Εαν ενα node ειναι στην κατασταση NEW, δημιουργει μια διεργασία-παιδι, αλλαζει την placeholder τιμη του PID με το PID της νεας διεργασιας, και αρχιζει να εκτελει την εντολη.
5. Εαν ενα node ειναι στην κατασταση STOPPED, σημαινει οτι υπαρχει ηδη διεργασία που εκτελει την εντολη, οποτε η διεργασία-γονεας της επιτρεπει να συνεχισει στελνοντας της ενα SIGCONT, και αλλαζει την κατασταση του node σε RUNNING.
6. Επειτα η διεργασία-γονεας μπαινει σε sleep για χρονο `time_slice` Ετσι :
 1. Στην περιπτωση του FCFS, η sleep θα επιστρεψει μετα απο τοσο πολυ χρονο, οπου με απολυτη σιγουρια θα εχει ολοκληρωθει η διεργασία και θα εχει ηδη στειλει SIGCHLD, αφυπνωντας την διεργασία-πατερα. Με αυτον τον τροπο ο κατα κορον Round Robin αλγοριθμος μπορει να εκτελεσει FCFS.
 2. Στην περιπτωση του RR, το sleep θα αφυπνησει την διεργασία-πατερα μετα απο το ορισμενο time slice, και εαν η διεργασία ολοκληρωθει νωριτερα, θα στειλει SIGCHLD, αφυπνιζοντας την διεργασία-πατερα.
7. Οταν ελευθερώνεται η διεργασία-γονεας, στελνει ενα SIGSTOP για να διακοψει την τυχον εκτελεση της διεργασιας-παιδι και ελεγχει εαν εχει ολοκληρωθει:
 1. Εαν εχει, υπολογιζεται ο χρονος που εχει περασει απο την αρχη εκτελεσης του προγραμματος, αλλαζει η κατασταση του node σε EXITED και τοποθετειται στην καινουργια ουρα, και εκτυπώνονται τα απαιρητητα στοιχεια.
 2. Εαν δεν εχει, αλλαζει την κατασταση της σε STOPPED, και την επανατοποθετει στην αρχικη ουρα.
8. Αφαιρείται το επομενο node απο την ουρα
9. Το loop ξεκινα ξανα.

10. Όταν το loop ολοκληρωθεί (δεν υπάρχουν διεργασίες που δεν έχουν τελειώσει), η συνάρτηση επιστρέφει έναν δείκτη στην ουρά με τις ολοκληρωμένες διεργασίες.

Παρατηρήσεις – προβλήματα:

Αρχικά είχαμε προσπαθήσει να τοποθετούμε τα ολοκληρωμένα nodes πίσω στον αρχικό πίνακα, και η συνάρτηση RR να είναι void. Αυτό μας προκαλούσε προβλήματα με το while loop (πχ κάποια διεργασία δεν θα ολοκληρωνε και θα ολοκληρωνε το πρόγραμμα), οπότε καταλήξαμε σε αυτή την πιο απλοϊκή υλοποίηση.

Παρατηρούμε πως το πρόγραμμα μοιάζει να λειτουργεί όπως θα έπρεπε, **καθώς** ο FCFS ολοκληρώνεται πιο γρήγορα απ'τον RR, πράγμα λογικό (ο RR έχει καθαρή σπαταλή χρόνου στις εναλλαγές διεργασιών), και πως ο FCFS καθυστερεί παρα πολύ να ολοκληρώσει πιο γρήγορες διεργασίες εάν αυτές βρίσκονται μετά από μεγάλες (reverse.txt).

Δευτερη φάση – scheduler_io.c

Για την δευτερη φάση, ο κώδικας είναι κατά βάση ο ίδιος (μαλιστα οι συναρτησεις είναι οι ιδιες).

Η διαφορά είναι ότι, καθώς τα προγράμματα που θα εκτελούν τα παιδια- διεργασίες θα στέλνουν στην διεργασία-πατέρα SIGUSR1 και SIGUSR2, πρέπει να τις συνδεσουμε με κάποιον handler, καθώς η default απάντηση σε ένα τέτοιο σήμα είναι ο τερματισμός της διεργασίας.

Δημιουργήσαμε λοιπόν δυο handlers, οι οποίοι μεταβάλλουν δυο παγκόσμιες μεταβλητές, την io_pending και την io_done (οταν ερχονται τα αντιστοιχα σηματα οι μεταβλητες γινονται 1).

Ουσιαστικά λειτουργούν σαν σημαφοροι για την αναγκη και την ολοκληρωση της αναγκης για i/o. Όταν η διεργασία παιδι στείλει το σήμα SIGUSR1 (που θα γίνει μέσα στο time slice της διεργασίας), το io_pending γίνεται 1, και λόγω του handler, η διεργασία-γονεας αφυπνίζεται και αμέσως διακοπεί την λειτουργία της διεργασίας.

Καθώς το io_pending είναι 1, η κατάσταση του node αλλάζει σε IO, και το πρόγραμμα συνεχίζει όπως στην προηγούμενη υλοποίηση (προφανώς εφόσον η διεργασία δεν έχει ολοκληρωθεί σίγουρα θα επανατοποθετηθεί στο τέλος της ουράς).

Όταν η διεργασία στείλει SIGUSR2 (κάτι που μπορεί να συμβεί μόνο στο time_slice της διεργασίας*), η μεταβλητή io_pending γίνεται 1, και πριν ξαναμπει στην ουρά η διεργασία, αυτή αλλάζει κατάσταση σε IO.

Όσον αφορά όλα τα υπολοιπα, λειτουργουν με τον ιδιο τροπο που επεξηγησαμε παραπανω.

Παρατηρησεις:

*Κανονικα, η διεργασία που ζηται i/o δεν θα επρεπε να συνεχισει, παρα μονο META που παραλαβει το απαιτητο i/o. Στην δικη μας περιπτωση, εαν η διεργασία ειναι σταματημενη, δεν θα παραλαβουμε ποτέ το SIGUSR2. Οποτε κανουμε αυτη την παραχωρηση, οπου το προγραμμα μπορει να τρεξει και με i/o.

Η υλοποιηση του FCFS σαν υποπεριπτωση του RR λειτουργει με πανομοιοτυπο τροπο με το προηγουμενο ερωτημα, ως RR με τεραστιο time slice, επιβεβαιωνοντας πως η διεργασία θα εχει τελειωσει πριν επελθει αυτη η στιγμη. Σε περιπτωση I/O, ο FCFS οπως και ο RR πρεπει να αλλαζει διεργασία, και αυτη ειναι η μονη περιπτωση στην οποια ο FCFS της υλοποιησης μας θα αλλαξει διεργασία πριν το περας της προηγουμενης (εκτος εαν η διεργασία παρει παραπανω απο 1000 δευτερολεπτα).

Παρατηρησαμε πως η εφαρμογη εχει τα ιδια αποτελεσματα εαν αντι για sleep(time_slice), βαλουμε μια εντολη alarm(time_slice) και μια εντολη pause() ακριβως απο κατω, εαν θεσουμε εναν κενο handler για το SIGALRM. Δεν χρησιμοποιησαμε αυτη την λογικη στο τελικο μας προγραμμα γιατι ηταν πολυ εξιδεικευμενη χωρις ιδιαιτερο λογο.

Επισης στην θεση του sleep θα μπορουσαμε να ειχαμε βαλει καποια αλλη παραλλαγη(βλ. usleep), αλλα καταληξαμε να χρησιμοποιούμε την πιο βασικη συναρτηση sleep, και να πολλαπλασιαζουμε τον αριθμο που μας εδωσε ο χρηστης με το 0.001.

Το αρχειο ./run.sh τρεχει σωστα και τις 4 ρουτινες.