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

Λειτουργουν ολα τα υποερωτηματα και στα 2 μερη.

Ο κωδικας μας αποτελείται απο 3 αρχεια.

Βοηθητικές συναρτησεις – queue (functions.c)

- Υλοποιήσαμε μια ουρά, η οποια κραταει στα nodes της το PID της διεργασίας που ειναι συνδεδεμενη μαζι της (καθε node αντιστοιχει σε μια εντολη που εκτελειται απο μια διεργασια-παιδι).
- Χρησιμοποιήσαμε την συναρτηση get_wtime(void) (που μας ειχατε δώσει στο πρώτο project) για χρονικους υπολογισμους.
- Για λογους ευκολιας, δημιουργησαμε μια συναρτηση που ελεγχει εαν η ουρα μας ειναι κενη.

Τα nodes περιεχουν πληροφοριες για το PID της διεργασιας, την εντολη που εκτελουν, την κατασταση της διεργασιας και εναν δεικτη στο επομενο node.

Πρώτη φαση – scheduler.c

Αφοτου αρχικοποιησουμε τα απαραιτητα (συμπεριλαμβανομενων του κενου handler για το σημα SIGCHLD και της παγκοσμιας μεταβλητης start_time), το πρωτο μας μελημα ειναι να διαβασουμε την εισοδο του χρηστη περι ειδος δρομολογησης, time_slice και αρχειου που περιεχει τις εφαρμογες. Αυτο γινεται ευκολο με αξιοποιηση του argv[].

Επειτα διαβαζουμε γραμμη γραμμη τις εντολες του αρχειου που μας ειχε υποδειξει ο χρηστης, και κανουμε enqueue με placeholder τιμη PID = 0 (η συσχετιση του Node με διεργασια θα γινει στην συναρτηση χρονοπρογραμματισμου), και αρχικο status NEW.

Στην συνεχεια, σε περιπτωση που ο χρηστης επιλεξε FCFS, καλειται η συναρτηση RR με time slice = 1000s (επεξηγηση για αυτη την επιλογη παρακατω), και εαν ο χρηστης επιλεξε RR, καλειται η συναρτηση RR με το time_slice που οριστηκε απο τον χρηστη.

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 διεργασια-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 διεργασια που εκτελει την εντολη, οποτε η διεργασια-γονεας της επιτρεπει να συνεχισει στελνοντας της ενα SIGCONT, και αλλαζει την κατασταση του node σε RUNNING.

6. Επειτα η διεργασια-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 διεργασια και θα εχει ηδη στειλει SIGCHLD, αφυπνωντας την διεργασια-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 διεργασια-πατερα μετα απο το ορισμενο time slice, και εαν η διεργασια ολοκληρωθει νωριτερα, θα στειλει SIGCHLD, αφυπνιζοντας την διεργασια-πατερα.

7. Οταν ελευθερώνεται η διεργασια-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 απαραιτητα 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. Οταν το loop ολοκληρωθει (δεν υπαρχουν διεργασιες που δεν εχουν τελειωσει), η συναρτηση επιστρεφει εναν δεικτη στην ουρα με τις ολοκληρωμενες διεργασιες.

Παρατηρησεις – προβληματα:

Αρχικά ειχαμε προσπαθησει να τοποθετουμε τα ολοκληρωμενα nodes πισω στον αρχικο πινακα, και η συναρτηση RR να ειναι void. Αυτο μας προκαλουσε προβληματα με το while loop (πχ καποια διεργασια δεν θα ολοκληρωνε και θα ολοκληρωνε το προγραμμα), οποτε καταληξαμε σε αυτη την πιο απλοικη υλοποιηση.

Παρατηρουμε πως το προγραμμα μοιαζει να λειτουργει οπως θα επρεπε, **καθως** ο FCFS ολοκληρωνεται πιο γρηγορα απ'τον RR, πραγμα λογικο (ο RR εχει καθαρη σπαταλη χρονου στις εναλλαγες διεργασιων), και πως ο FCFS καθυστερει παρα πολυ να ολοκληρωσει πιο γρηγορες διεργασιες εαν αυτες βρισκονται μετα απο μεγαλες (reverse.txt).

Δευτερη φαση – scheduler_io.c

Για την δευτερη φαση, ο κωδικας ειναι κατα βαση ο ιδιος (μαλιστα οι συναρτησεις ειναι οι ιδιες).

Η διαφορά ειναι οτι, καθως τα προγραμματα που θα εκτελουν τα παιδια-διεργασιες θα στελνουν στην διεργασια-πατερα SIGUSR1 και SIGUSR2, πρεπει να τις συνδεσουμε με καποιον handler, καθως η default απαντηση σε ενα τετοιο σημα ειναι ο τερματισμος της διεργασιας.

Δημιουργησαμε λοιπον δυο handlers, οι οποιοι μεταβαλλουν δυο παγκοσμιες μεταβλητές, την io_pending και την io_done (οταν ερχονται τα αντιστοιχα σηματα οι μεταβλητες γινονται 1).

Ουσιαστικά λειτουργουν σαν σημαφοροι για την αναγκη και την ολοκληρωση της αναγκης για i/o. Οταν η διεργασια παιδι στειλει το σημα SIGUSR1 (που θα γινει μεσα στο time slice της διεργασιας), το io_pending γινεται 1, και λογω του handler, η διεργασια-γονεας αφυπνιζεται και αμεσως διακοπτει την λειτουργια της διεργασιας.

Καθως το io_pending ειναι 1, η κατασταση του node αλλαζει σε IO, και το προγραμμα συνεχιζει οπως στην προηγουμενη υλοποιηση *(προφανως εφοσον η διεργασια δεν εχει ολοκληρωθει σιγουρα θα επανατοποθετηθει στο τελος της ουρας)*.

Οταν η διεργασια στειλει SIGUSR2 *(κατι που μπορει να συμβει μονο στο time_slice της διεργασιας*)*, η μεταβλητη io_pending γινεται 1, και πριν ξαναμπει στην ουρα η διεργασια, αυτη αλλαζει κατασταση σε IO.

Οσον αφορα ολα τα υπολοιπα, λειτουργουν με τον ιδιο τροπο που επεξηγησαμε παραπανω.

Παρατηρησεις:

*Κανονικα, η διεργασια που ζηταει i/o δεν θα επρεπε να συνεχισει, παρα μονο ΜΕΤΑ που παραλαβει το απαραιτητο i/o. Στην δικη μας περιπτωση, εαν η διεργασια 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 διεργασια θα εχει τελειωσει πριν επελθει αυτη η στιγμη. Σε περιπτωση I/O, ο FCFS οπως και ο RR πρεπει να αλλαζει διεργασια, και αυτη ειναι η μονη περιπτωση στην οποια ο FCFS της υλοποιησης μας θα αλλαξει διεργασια πριν το περας της προηγουμενης (εκτος εαν η διεργασια 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.