Τ.Ε.Ι ΠΕΛΟΠΟΝΝΗΣΟΥ

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ (έδρα: Σπάρτη)

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ Τ.Ε.

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Σχεδίαση και ανάπτυξη συστήματος προσομοίωσης χρονοδρομολόγησης, βάσει προτεραιοτήτων, με εξαρτήσεις μεταξύ των διεργασιών

Της φοιτήτριας:

Φιλανδριανού Χρυσούλας

Επιβλέπων καθηγητής:

Δρ. Μάργαρης Διονύσιος,
Επιστημονικός Συνεργάτης
ΤΕΙ Πελοποννήσου

2018

ΔΗΛΩΣΗ ΜΗ ΛΟΓΟΚΛΟΠΗΣ ΚΑΙ ΑΝΑΛΗΨΗΣ ΠΡΟΣΩΠΙΚΗΣ ΕΥΘΥΝΗΣ

"Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ενυπογράφως ότι είμαι αποκλειστικός συγγραφέας της παρούσας Πτυχιακής Εργασίας, για την ολοκλήρωση της οποίας κάθε βοήθεια είναι πλήρως αναγνωρισμένη και αναφέρεται λεπτομερώς στην εργασία αυτή. Έχω αναφέρει πλήρως και με σαφείς αναφορές, όλες τις πηγές χρήσης δεδομένων, απόψεων, θέσεων και προτάσεων, ιδεών και λεκτικών αναφορών, είτε κατά κυριολεξία είτε βάση επιστημονικής παράφρασης. Αναλαμβάνω την προσωπική και ατομική ευθύνη ότι σε περίπτωση αποτυχίας στην υλοποίηση των ανωτέρω δηλωθέντων στοιχείων, είμαι υπόλογος έναντι λογοκλοπής, γεγονός που σημαίνει αποτυχία στην Πτυχιακή μου Εργασία και κατά συνέπεια αποτυχία απόκτησης του Τίτλου Σπουδών, πέραν των λοιπών συνεπειών του νόμου περί πνευματικών δικαιωμάτων. Δηλώνω, συνεπώς, ότι αυτή η Πτυχιακή Εργασία προετοιμάστηκε και ολοκληρώθηκε από εμένα προσωπικά και αποκλειστικά και ότι, αναλαμβάνω πλήρως όλες τις συνέπειες του νόμου στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δε μου ανήκει διότι είναι προϊόν λογοκλοπής άλλης πνευματικής ιδιοκτησίας."

Όνομα και Επώνυμο Συγγραφέα (Με Κεφαλαία):

ΧΡΥΣΟΥΛΑ ΦΙΛΑΝΔΡΙΑΝΟΥ

Υπογραφή (Ολογράφως, χωρίς μονογραφή):

Ημερομηνία (Ημέρα – Μήνας – Έτος):

22 ΜΑΡΤΙΟΥ 2018

Ευχαριστίες

Ευχαριστίες, ίσως η πιο άβολη στιγμή της πτυχιακής, όταν αυτούς που θεωρείς υπεύθυνους για την ολοκλήρωση ενός κύκλου πρέπει να τους απαριθμήσεις και να μην ξεχάσεις κανέναν. Σφραγίζοντας, λοιπόν, το κύκλο των σπουδών με την πτυχιακή αυτή. Τέσσερα υπέροχα χρόνια και κάτι, τα φοιτητικά χρόνια, μόλις έφτασαν στο τέλος τους και οφείλω ένα ευχαριστώ πρώτα από όλα, στους γονείς μου που μου έδωσαν τη δυνατότητα να σπουδάσω και τη γιαγιά μου που όλα τα χρόνια μου αφιέρωσε άπειρες ώρες διαβάσματος. Μετά να ευχαριστήσω όλους τους καθηγητές και δασκάλους μου που με ανέχτηκαν και πάντα έβρισκαν τον τρόπο να μου κεντρίζουν το ενδιαφέρον και να με κάνουν να προσπαθώ παραπάνω. Μα πάνω από όλα να ευχαριστήσω τον επιβλέποντα καθηγητή μου, που μου αφιέρωσε όσο χρόνο και προσοχή χρειαζόμουν και με καθοδήγησε για την ολοκλήρωση της εργασίας αυτής. Άδικο θα ήταν να μην αναφέρω, αυτόν τον κύριο « Εύδοξο» που κάθε χρόνο μου παρείχε όλα τα βιβλία που χρειαζόμουν και τώρα γλίτωσα πολύτιμο χρόνο έχοντας στη κατοχή μου όλα τα απαραίτητα βιβλία για την συγγραφή, όπως του κύριου «Tanenbaum», όχι ότι το διαδίκτυο δεν ήταν πάντα δίπλα μου να με στηρίζει και να με γεμίζει με ιδέες και γνώσεις. Και αφού ανέφερα, νομίζω, όλους πρέπει να αναφέρω τα δύο «Φ» της ζωής μου που έχουν ακούσει τη περισσότερη γκρίνια όλα αυτά τα χρόνια, το παππού και την αδερφή μου.

Σας ευχαριστώ πολύ όλους για την υπομονή και την αγάπη σας.

Περιεχόμενα

Εισαγωγή

Η πτυχιακή εργασία τοποθετείται επιστημονικά στο αντικείμενο των Λειτουργικών Συστημάτων.

Σκοπός της πτυχιακής αυτής, είναι η σχεδίαση και ανάπτυξη συστήματος προσομοίωσης χρονοδρομολόγησης με δοσμένες τιμές παραμέτρων (χρόνος εισαγωγής, χρόνος εκτέλεσης, προτεραιότητα, λειτουργία Down σε σημαφόρο του συστήματος, κλπ), τόσο χωρίς όσο και με εξαρτήσεις μεταξύ των διεργασιών (με τη χρήση σημαφόρων).

Αρχικά, η πτυχιακή εργασία εισάγει τον αναγνώστη στο αντικείμενο των λειτουργικών συστημάτων, αναφέροντας και εξηγώντας χρήσιμες έννοιες, για την κατανόησή της.

Στη συνέχεια επικεντρώνεται στο αντικείμενο του χρονοπρογραμματισμού διεργασιών, δηλαδή, της δημιουργίας της αλληλουχίας εκτέλεσης των διεργασιών από το Λειτουργικό Σύστημα, όπου και αναλύονται οι βασικές πολιτικές χρονοπρογραμματισμού.

Έπειτα, παρουσιάζονται τμήματα του κώδικα, που υλοποιήθηκε, για την προσομοίωση. Η ανάπτυξη του κώδικα έχει γίνει σε δύο σκέλη. Στο πρώτο σκέλος υλοποιήθηκαν όλες οι βασικές πολιτικές χρονοπρογραμματισμού, ενώ στο δεύτερο σκέλος αναπτύχθηκε κώδικας για την υποστήριξη εξαρτήσεων μεταξύ διεργασιών, με τη χρήση σημαφόρων.

Τέλος παρουσιάζονται τα αποτελέσματα των εκτελέσεων (ως στατιστικά στοιχεία) και τα συμπεράσματα της πτυχιακής αυτής εργασίας, τα οποία επαληθεύουν τη θεωρία.

Κεφάλαιο 1 - Βασικές Έννοιες Λειτουργικών Συστημάτων

Σε αυτό το εισαγωγικό κεφάλαιο θα αναλυθούν βασικές έννοιες των Λειτουργικών Συστημάτων, καθώς και ο τρόπος υλοποίησής τους. Μια καλή κατανόηση αυτών των εννοιών είναι σημαντική για την κατανόηση των θεμάτων που παρουσιάζονται στα επόμενα κεφάλαια. Περιληπτικά το κεφάλαιο αυτό θα ασχοληθεί με τις βασικές λειτουργίες και τους στόχους των Λειτουργικών Συστημάτων, τη δομή τους, και θα αναλυθούν έννοιες όπως,της διεργασίας, της διεργασιακής επικοινωνίας, τον ρόλο των σημαφόρων και τον τρόπο διαχείρισής τους.

1.1 Λειτουργικό Σύστημα

Προτού περιγραφούν ζητήματα σχετικά με την υλοποίηση του αλγορίθμου χρονοπρογραμματισμού ενός Λειτουργικού Συστήματος, είναι αναγκαία η κατανόηση μερικώνεννοιών και σημαντικών ορισμών σε σχέση με το Λειτουργικό Σύστημα.

Ορισμός:

Λειτουργικό Σύστημα (Λ.Σ.) ή **Operating System (O.S.)** ονομάζεται το λογισμικό του υπολογιστή που είναι υπεύθυνο για τη διαχείριση και τον συντονισμό των εργασιών, καθώς και την κατανομή των διαθέσιμων πόρων.
Το λειτουργικό σύστημα παρέχει ένα θεμέλιο, ένα μεσολαβητικό επίπεδο λογικής διασύνδεσης μεταξύ λογισμικού και υλικού,διαμέσου του οποίου οι εφαρμογές αντιλαμβάνονται εμμέσως τον υπολογιστή.
Μια από τις κεντρικές αρμοδιότητες του λειτουργικού συστήματος είναι η διαχείριση του υλικού, απαλλάσσοντας έτσι το λογισμικό του χρήστη από τον άμεσο και επίπονο χειρισμό του υπολογιστή και καθιστώντας ευκολότερο τον προγραμματισμό τους [5].

Το λειτουργικό σύστημα περιέχει κώδικα χαμηλού επιπέδου αποκλειστικά για την αρχιτεκτονική του επεξεργαστή στην οποία εκτελείται, γραμμένο είτε σε υψηλού επιπέδου γλώσσα (C), είτε απευθείας σε συμβολική γλώσσα. Όλοι οι υπολογιστές κάνουν χρήση ενός λειτουργικού συστήματος, στο παρελθόν παρατηρείται ότι βασίζονταν σε ενσωματωμένο λειτουργικό, το οποίο παρέχονταν με κάποιο οπτικό μέσο ή κάποια συσκευή αποθήκευσης.

Υπηρεσίες Λειτουργικού Συστήματος

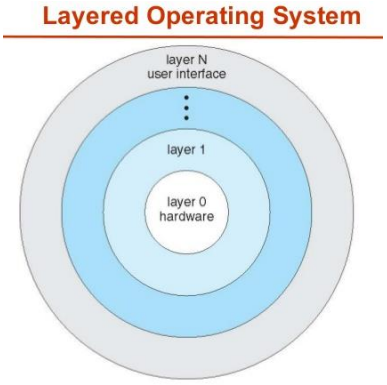
Ένα λειτουργικό σύστημα παρέχει κάποιες σημαντικές υπηρεσίες στον χρήστη και στο σύστημα γενικότερα. Οι πιο σημαντικές είναι:

Διαχείριση Διεργασιών
- **Δημιουργία και καταστολή διεργασιών**
- **Επικοινωνία και συγχρονισμός διεργασιών**
Διαχείριση Μνήμης
- **Διαχείριση μνήμης ανάμεσα σε πολλαπλές διεργασίες**
- **Ανάθεση μνήμης κατόπιν αίτηση από μία διεργασία**
Διαχείριση Αρχείων
- **Δημιουργία και επεξεργασία αρχείων και καταλόγων**
Διαχείριση Συστήματος Εισόδου / Εξόδου (Ε/Ε)
- **Διαχείριση συσκευών εισόδου / εξόδου**
Διαχείριση Συσκευών Δευτερεύουσας Αποθήκευσης
- **Διαχείριση ελεύθερου χώρου σε μονάδες δίσκων**
Δικτυακή Υποστήριξη
- **Παροχή TCP/IP και σχετικών πρωτοκόλλων (HTTP, FTP, SSL)**
Προστασία και Ασφάλεια
- **Παροχή πιστοποίησης**
- **Προστασία από κακόβουλες επιθέσεις μέσω δικτύου**

Πίνακας 1.1 – Υπηρεσίες Λειτουργικών Συστημάτων [1]

Δομή Λειτουργικών Συστημάτων

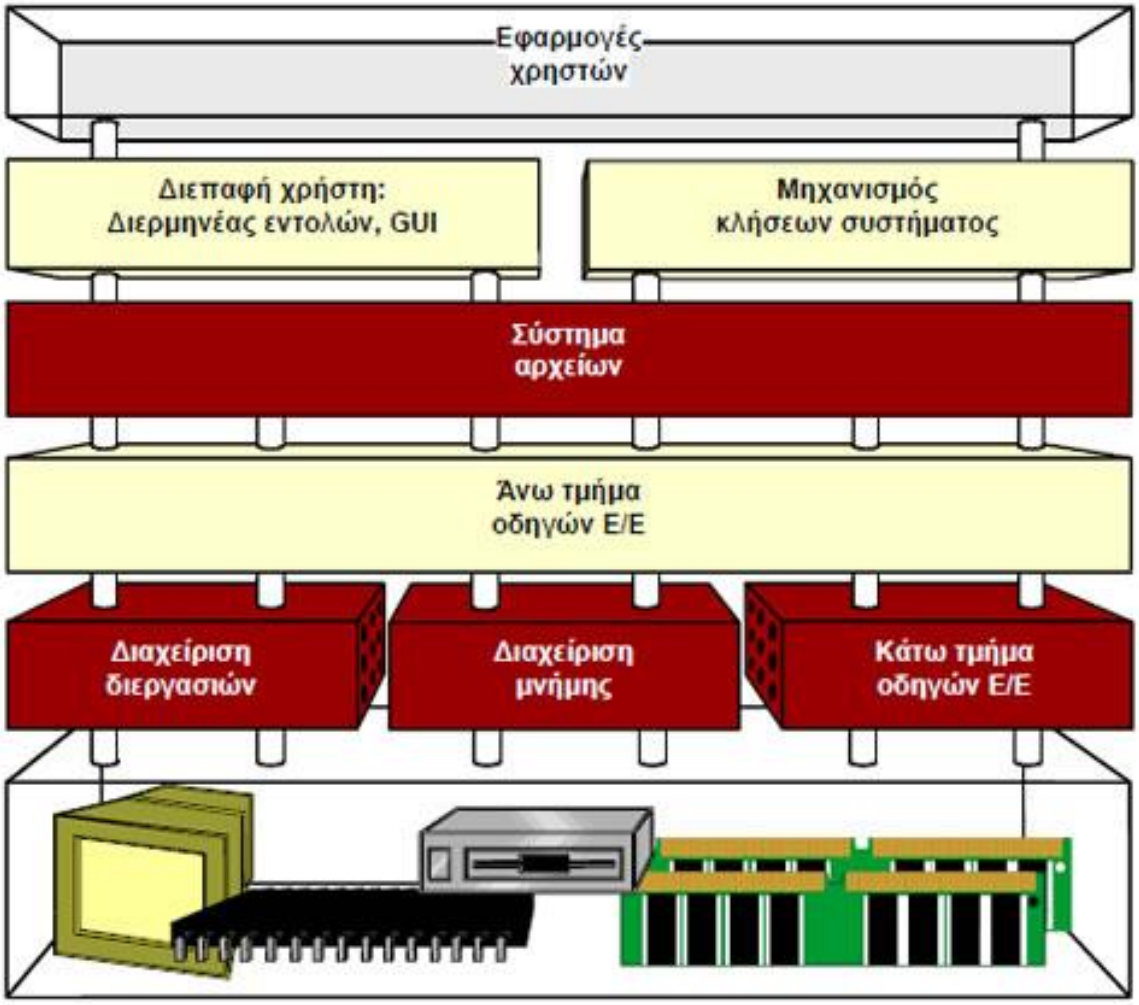
Η συνηθέστερη δομή ενός Λειτουργικού Συστήματος είναι σε επίπεδα (layers), που σημαίνει ότι η σχεδίαση τους έχει διαιρεθεί σε τμήματα και κάθε τμήμα επικοινωνεί μόνο με αυτά που βρίσκονται στο αμέσως ανώτερο ή κατώτερο επίπεδο από αυτό. Όσα τμήματα κάνουν χρήση το υλικό του υπολογιστή , απευθείας, βρίσκονται στο κατώτερο επίπεδο του Συστήματος, το οποίο ονομάζεται layer 0 και είναι το hardware, σε αντίθεση με το ανώτερο – layerN– που είναι το userinterface (διεπαφή του χρήστη). Όπως φαίνεται στην εικόνα.



Εικόνα 1.1 Στρώματα Λειτουργικού Συστήματος

Τα προγράμματα των χρηστών επικοινωνούν μόνο με το υψηλότερο επίπεδο, και η επικοινωνία αυτή επιτυγχάνεται με τρείς τρόπους: είτε με προγράμματα γραμμένα από τον ίδιο τον χρήστη, είτε με εντολές μέσω command interpreter , shell (διερμηνέα εντολών) ή με χρήση γραφικών γνωστό με το όνομα GUI (Graphical User Interface).

Το πλεονέκτημα αυτής της δομής είναι ο αρθρωτός (modular) σχεδιασμός που έχει ως αποτέλεσμα τον ευκολότερο σχεδιασμό του λειτουργικού συστήματος. Το βασικό πρόβλημα είναι ο σωστός διαχωρισμός λειτουργιών ανά επίπεδο.Στη παρακάτω εικόνα φαίνεται σχηματικά μία ενδεικτική δομή:



Εικόνα 1.2 Δομή λειτουργικού συστήματος

Σε αυτό το σημείο σκόπιμο θα είναι ο ορισμός των βασικών παραγόντων υλοποίησης ενός Λειτουργικού συστήματος:

- Καταστάσεις του επεξεργαστή

- Πυρήνας
- Μέθοδος αίτησης υπηρεσίας του συστήματος

Λίγα λόγια για το κάθε παράγοντα. Οι καταστάσεις του επεξεργαστή (CPU) είναι δύο:

- όταν μια διεργασία εκτελείται σε κατάσταση χρήστη και δεν έχει πρόσβαση σε μνήμη πέρα από το δικό της χώρο μνήμης,
- όταν εκτελείται σε κατάσταση πυρήνα όπου μπορεί να αναφερθεί σε οποιοδήποτε μέρος της μνήμης.

Ο πυρήνας (kernel) βρίσκεται φορτωμένος στη μνήμη του υπολογιστή διαρκώς και αναλαμβάνει τη διανομή της μνήμης μεταξύ εφαρμογών (memorymanagement) , του χρόνου μεταξύ διεργασιών (process scheduling) και περιλαμβάνει οδηγούς συσκευών (device drivers), όπως δίσκους, κάρτες δικτύου, σειριακές θύρες κ.α., πρωτόκολλα επικοινωνίας, υποστήριξη συστημάτων αρχείου (file systems) και άλλο κώδικα για τη διαχείριση του λειτουργικού συστήματος και του υπολογιστή. Τέλος οι μέθοδοι υπηρεσίας συστήματος εκτελούνται μέσω και των δύο καταστάσεων επεξεργαστή που αναφέρθηκαν παραπάνω και υπάρχουν δύο τεχνικές:

1) κλήσεις συστήματος, δηλαδή η διεπαφή ανάμεσα σε ένα πρόγραμμα χρήστη και το λειτουργικό σύστημα,

2) πέρασμα / ανταλλαγή μηνυμάτων, δηλαδή η δημιουργία του κατάλληλου μηνύματος για την επιτυχημένη επικοινωνία ανάμεσα σε μία εφαρμογή χρήστη και στον πυρήνα.

1.2 Διεργασία (Process)

Διεργασία ή ακολουθιακή διεργασία (sequentialprocess), μία από τις πιο σημαντικές έννοιες των Λειτουργικών Συστημάτων, πρόκειται για ένα εκτελέσιμο πρόγραμμα που υποστηρίζει τη δυνατότητα ύπαρξης (ψευδο-) ταυτόχρονης λειτουργίας όταν υπάρχει διαθέσιμη μόνο μία CPU, μετατρέποντάς τη σε πολλές εικονικές, αναγκάζοντας τη CPU να εναλλάσσεται από πρόγραμμα σε πρόγραμμα – από διεργασία σε διεργασία. Η εναλλαγή αυτή, ονομάζεται πολυπρογραμματισμός (multi-programming).[6]

Η κάθε διεργασία συνοδεύεται από το χώρο διευθύνσεων (address space), που περιέχει το εκτελέσιμο πρόγραμμα, τα δεδομένα του προγράμματος και τη στοίβα του,είναι δηλαδή, μια λίστα από θέσεις μνήμης που μεταβάλλονται από το ελάχιστο όριο έως και το μέγιστο, σχηματίζοντας μία περιοχή μέσα στην οποία η διεργασία μπορεί να διαβάσει και να γράψει. Επίσης, η κάθε διεργασία είναι συνδεδεμένη με ένα σύνολο πόρων στους οποίους περιλαμβάνονται οι καταχωρητές, δηλαδή ένα σύνολο ανοιχτών

αρχείων – εκκρεμών προειδοποιήσεων – λιστών συναφών διεργασιών – απαραίτητων πληροφοριών για την εκτέλεση του προγράμματος.

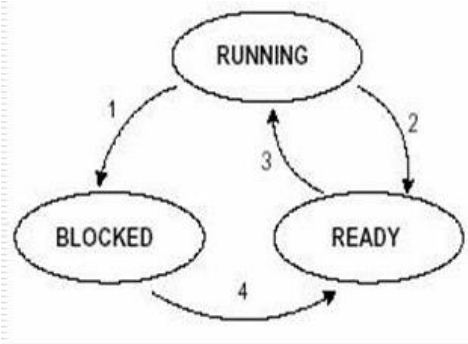
Καταστάσεις Διεργασιών

Κάθε διεργασία όπως αναφέρεται παραπάνω είναι μία ανεξάρτητη οντότητα, αφού διαθέτει το δικό της μετρητή προγράμματος και τη δική της εσωτερική κατάσταση, όμως συχνά απαιτείται η αλληλεπίδραση της με άλλες διεργασίες. Δηλαδή, η είσοδος μίας διεργασίας μπορεί να είναι η έξοδος μίας άλλης ή για την ολοκλήρωση μίας διεργασίας μπορεί να χρειάζεται η προσωρινή της αναστολή και ολοκλήρωση μίας άλλης. Για αυτό το λόγο υπάρχουν τρεις δυνατές καταστάσεις στις οποίες μπορεί να βρεθεί μία διεργασία:

a. Εκτελούμενη (running), πραγματική χρήση της CPU τη δεδομένη στιγμή.

b. Έτοιμη (ready), ή εκτελέσιμη (runnable), έχει διακοπεί προσωρινά για να εκτελεστεί μία άλλη διεργασία.

c. Μπλοκαρισμένη (blocked), δεν μπορεί να συνεχίσει την εκτέλεσή της μέχρι να λάβει χώρα από κάποιο εξωτερικό συμβάν.



Εικόνα 1.3 Κύκλος καταστάσεων διεργασιών

Η βασική διαφορά των δύο πρώτων καταστάσεων (1 και 2) είναι ότι η δεύτερη δεν είναι προσωρινά διαθέσιμη, ενώ προσπαθούν να εκτελεστούν. Η τρίτη κατάσταση (3) είναι διαφορετική από τις προηγούμενες αφού δεν μπορεί να εκτελεστεί όσο η CPUείναι απασχολημένη με κάποια άλλη.

Οι βασικές μεταβάσεις μεταξύ αυτών των τριών καταστάσεων είναι τέσσερις:

1. Running to blocked – Η διεργασία μπλοκάρεται καθώς περιμένει δεδομένα εισόδου.
2. Running to Ready – Ο χρονοπρογραμματιστής επιλέγει άλλη διεργασία.

3. Ready to Running – Ο χρονοπρογραμματιστής επιλέγει τη συγκεκριμένη διεργασία.
4. Blocked to Ready – Τα δεδομένα εισόδου είναι διαθέσιμα.

Πιο αναλυτικά, η πρώτη μετάβαση (1) συμβαίνει όταν διαπιστωθεί από το πρόγραμμα ότι δεν μπορεί να συνεχίσει να εκτελείται. Η δεύτερη και τρίτη μετάβαση (2 και 3) προκαλούνται από το χρονοπρογραμματιστή διεργασιών (process scheduler), ο οποίος αποτελεί τμήμα του λειτουργικού συστήματος. Η μετάβαση από την κατάσταση «εκτελούμενη» (running) στη κατάσταση «έτοιμη» (ready), μετάβαση 2, συμβαίνει όταν ο χρονοπρογραμματιστής αποφασίσει ότι η διεργασία έχει εκτελεστεί για μεγάλο χρονικό διάστημα και είναι η κατάλληλη στιγμή να εκτελεστεί κάποια άλλη διεργασία στη CPU. Η τρίτη μετάβαση (3) εμφανίζεται όταν όλες οι διεργασίες έχουν ολοκληρώσει όλο το χρόνο που τους αντιστοιχεί στη CPU και τότε είναι η κατάλληλη στιγμή να παραχωρηθεί στη συγκεκριμένη διεργασία ο χρόνος που της αναλογεί στον επεξεργαστή για να εκτελεστεί με τη σειρά της. Να σημειωθεί στο συγκεκριμένο σημείο ότι, καθοριστικό ρόλο για το χρόνο και τη σειρά των διεργασιών στη CPU, έχει ο αλγόριθμος χρονοπρογραμματισμού που θα αναλύσουμε παρακάτω. Η τέταρτη μετάβαση, από τη κατάσταση «μπλοκαρισμένη» (blocked) στη κατάσταση «έτοιμη» (ready), παρουσιάζεται όταν πραγματοποιείται ένα εξωτερικό συμβάν, όπως η ολοκλήρωση μίας άλλης διεργασίας και το αποτέλεσμά αυτής δοθεί ως είσοδο σε αυτή, τότε πραγματοποιείται αυτή η μετάβαση και αν δεν εκτελείται κάποια άλλη διεργασία παρατηρείται μία ακόμα μετάβαση, η μετάβαση 2, δηλαδή από τη κατάσταση «εκτελούμενη» (running) στη κατάσταση «έτοιμη» (ready).[4]

Γενικά, οι διεργασίες δεν εκτελούνται μεμονωμένες από το υπόλοιπο σύστημα, όπως φαίνεται και από τη τελευταία μετάβαση. Υπάρχει ανάγκη ανταλλαγής δεδομένων ανάμεσα στις διεργασίες για την εκτέλεσή τους ή για τον συγχρονισμό τους. Η επικοινωνία αυτή ονομάζεται Διεργασιακή επικοινωνία (InterprocessCommunication – IPC) και οι τεχνικές για την επίτευξη αυτής της επικοινωνίας μπορεί να είναι μέσω:

1. Διοχέτευσης εισόδου / εξόδου (Pipes)
2. Ουρών μηνυμάτων (Message Queues)
3. Διαμοιραζόμενης μνήμης (Shared Memory)
4. Υποδοχών (Sockets)
5. Σηματοφόρων ή Σημαφόρων (Semaphores)

Μπλοκ Ελέγχου Διεργασίας

Το Μπλοκ Ελέγχου Διεργασίας (Process Control Block – PCB), είναι μια δομή, όπου για κάθε ενεργή διεργασία, το λειτουργικό σύστημα αποθηκεύει πληροφορία για

αυτή, όπως η κατάστασή της, το id της (pid), το σημείο που βρίσκεται η εκτέλεσή της, κλπ, προκειμένου να μπορεί ομαλά να αλλάζει δυναμικά τη σειρά εκτέλεσης διεργασιών (context switching).[3]

1.3 Σηματοφόρος ή Σημαφόρος (Semaphore)

Το 1965, ο E.W. Dijkstra εισήγαγε μία νέα μεταβλητή για την καταμέτρηση των σημάτων αφύπνισης που έχουν αποθηκευτεί για μελλοντική χρήση, ονομάζεται σημαφόρος ή σηματοφόρος, και έτσι βρέθηκε μία γενική λύση για το πρόβλημα του συγχρονισμού των παράλληλων διεργασιών. Η μεταβλητή αυτή παίρνει τη τιμή 0, όταν δεν έχουν αποθηκευτεί σήματα αφύπνισης, και κάποιο θετικό ακέραιο ίσο με τις εκκρεμείς αφυπνίσεις.[2]

Ένας σημαφόρος έχει δύο λειτουργίες:

1. Down (sleep)
2. Up (wakeup)

Στη πρώτη λειτουργία αφαιρείται 1 από το μετρητή του σημαφόρου, ενώ στη δεύτερη προστίθεται 1.

Οι δύο κατηγορίες των σημαφόρων:

1. Γενικοί σημαφόροι (generalsemaphores), όπου ο μετρητής των θετικών ακέραιων τιμών αυξάνεται κανονικά.
2. Δυαδικοί σημαφόροι (binarysemaphores), όπου οι τιμές που παίρνει ο σημαφόρος είναι 0 ή 1.

Κάθε πρόγραμμα για να χειριστεί σημαφόρους θα πρέπει να συμπεριλάβει τις παρακάτω βιβλιοθήκες/αρχεία:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

Εικόνα 1.4 Βιβλιοθήκες σημαφόρων

1.4 Χρονοπρογραμματισμός Διεργασιών

Επανειλημμένα αναφέρεται ο όρος «χρονοπρογραμματισμός» και «πολυπρογραμματισμός», εδώ θα περιγραφεί η σημασία της ύπαρξης του όρου αυτού. Όταν ένας υπολογιστής είναι πολυπρογραμματισμένος, πολλές διεργασίες διεκδικούν τον έλεγχο της CPU, αυτή η διεκδίκηση συμβαίνει ανάμεσα στις διεργασίες που είναι έτοιμες να εκτελεστούν. Αν υπάρχει μόνο μία CPU, τότε θα πρέπει να ληφθεί απόφαση για το ποια διεργασία θα εκτελεστεί πρώτη, και τότε υπεύθυνο για αυτή την απόφαση είναι το λειτουργικό σύστημα. Το τμήμα του λειτουργικού συστήματος που είναι υπεύθυνο για την επιλογή αυτή ονομάζεται χρονοπρογραμματιστής (scheduler) ενώ ο αλγόριθμος που χρησιμοποιείται λέγεται αλγόριθμος χρονοπρογραμματισμού (scheduling algorithm). Οι βασικοί στόχοι του αλγορίθμου χρονοπρογραμματισμού [3] βάση συνθηκών είναι:

Για όλα τα συστήματα:

1. Δικαιοσύνη – κάθε διεργασία δικαιούται ίσο μερίδιο χρόνου από τον επεξεργαστή.
2. Επιβολή πολιτικής – η παρακολούθηση αν εκτελείται η καθορισμένη πολιτική.
3. Ισορροπία – διατήρηση σε κατάσταση «ενεργή» όλων των τμημάτων του συστήματος.

Συστήματα δέσμης:

1. Διεκπεραιωτική ικανότητα – η μεγιστοποίηση του αριθμού διεργασιών που επεξεργάζεται η CPU ανά ώρα.
2. Χρόνος διεκπεραίωσης – η ελαχιστοποίηση του χρόνου που μεσολαβεί ανάμεσα στην υποβολή και την ολοκλήρωση εργασίας.
3. Αξιοποίηση της CPU – διατήρηση της CPU σε ενεργή κατάσταση.

Αλληλεπιδραστικά συστήματα:

1. Χρόνος απόκρισης – η απόκριση στις αιτήσεις να είναι ταχύτατη.
2. Τήρηση αναλογιών – ικανοποίηση των προσδοκιών των χρηστών.

Συστήματα πραγματικού χρόνου:

1. Τήρηση προθεσμιών – αποφυγή απωλειών δεδομένων.
2. Προβλεψιμότητα – αποφυγή υποβιβασμού της ποιότητας στο σύστημα.

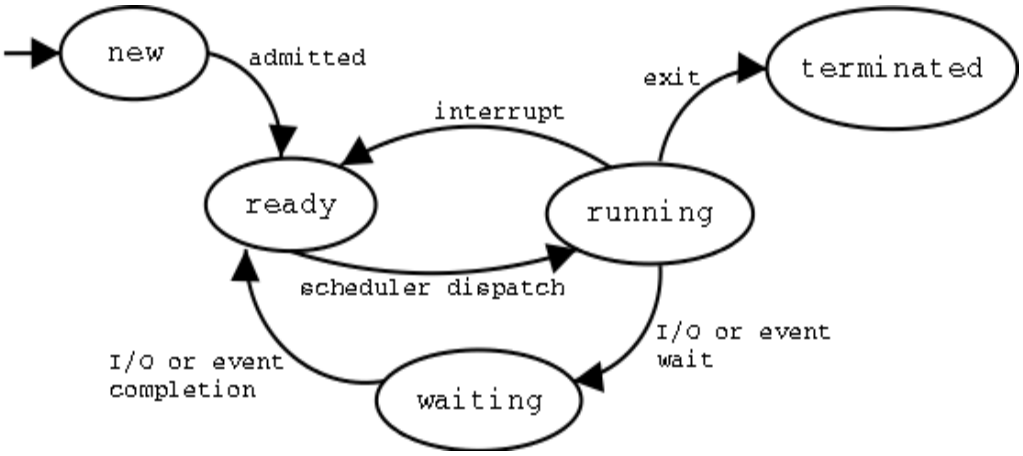
Ο αλγόριθμος είναι υπεύθυνος για την επιλογή ποιας διεργασίας και για πόσο χρονικό διάστημα θα απασχολεί τον επεξεργαστή, δηλαδή με μία λέξη η χρονοδρομολόγησή της.

Οι χρονοδρομολογητές γενικότερα έχουν απασχολήσει τους επιστήμονες αρκετά για αρκετούς λόγους, όπως, για τη συμπεριφορά τους που είναι κρίσιμη για την επίδοση κάθε διεργασίας αλλά, και για ολόκληρη τη συμπεριφορά του συστήματος. Γενικά, μελετήθηκαν διαφορετικές μεθοδολογίες που βοήθησαν στην έρευνα του λειτουργικού συστήματος. Τα προβλήματα των χρονοδρομολογητών κατάφεραν να κεντρίσουν το ενδιαφέρον της επιστημονικής κοινότητας.

Μετά από πολύ έρευνα αποδείχθηκε ότι οι αποφάσεις του χρονοδρομολογητή κατηγοριοποιούνται σε τέσσερις περιπτώσεις:

1. Όταν μια διεργασία αλλάζει από εκτελέσιμη κατάσταση (running) σε κατάσταση αναμονής (waiting) (I/O ή event waiting).
2. Όταν μια διεργασία αλλάζει από εκτελέσιμη κατάσταση (running) σε κατάσταση ετοιμότητας (ready) (για παράδειγμα αν συμβεί μία διακοπή (interrupt)).
3. Όταν μια διεργασία αλλάζει από κατάσταση αναμονής (waiting) σε κατάσταση ετοιμότητας (ready) (για παράδειγμα ολοκλήρωση διαδικασίας εισόδου / εξόδου (I/O) ή eventcompletion).
4. Όταν μια διεργασία από κατάσταση «εκτελέσιμη» (running) σε κατάσταση «εξόδου» (terminated ή exit).

Για την καλύτερη κατανόηση βοηθά η παρακάτω εικόνα:



Εικόνα 1.5 Διάγραμμα Καταστάσεων Διεργασιών (Transition State Diagram)

Στις περιπτώσεις 1 και 4 λέμε ότι έχουμε μη προεκχωρητικό χρονοπρογραμματισμό (non-preemptive scheduling), δηλαδή μια διεργασία δεσμεύει τον επεξεργαστή μέχρι να ολοκληρωθεί και μετά βάση του αλγόριθμου επιλέγεται η επόμενη που θα εκτελεστεί,

ενώ στις περιπτώσεις 2 και 3 έχουμε χρονοπρογραμματισμό προεκχώρησης (preemptivescheduling), που σημαίνει ότι αναλόγως του αλγόριθμου που έχει επιλεχθεί μια διεργασία μπορεί να διακόψει την εκτελέσιμη και να την βάλει σε κατάσταση αναμονής (waiting).

Οι πιο γνωστές μέθοδοι χρονοπρογραμματισμού είναι:

1. First Come First Served (FCFS)
2. Shortest job First (SJF)
3. Priority Scheduling (PR)
4. Round Robin (RR)

Κεφάλαιο 2 – Υλοποίηση Μεθόδων Χρονοπρογραμματισμού

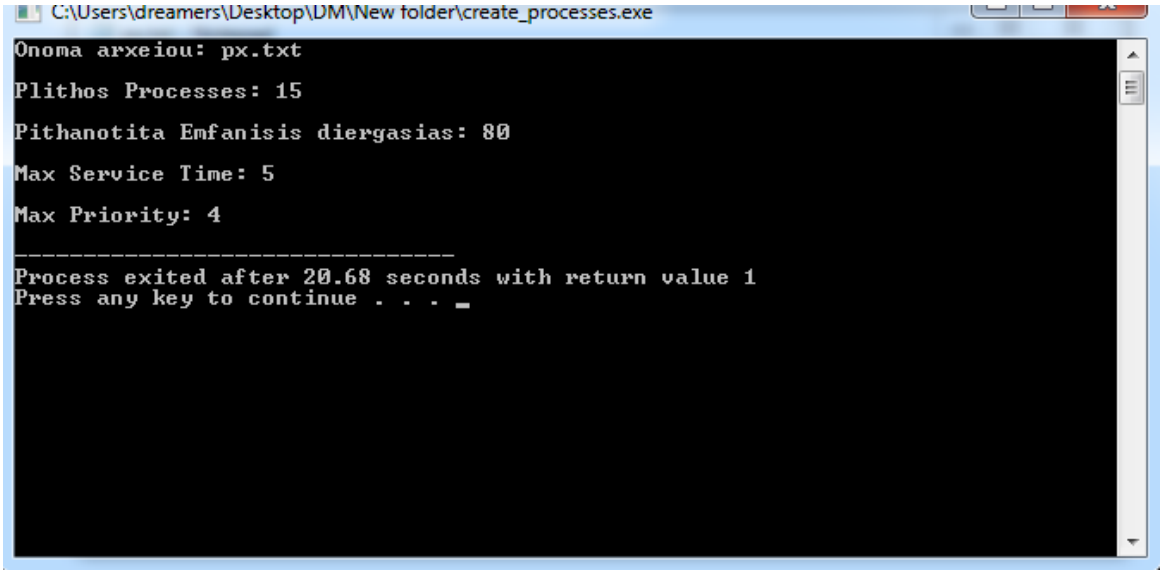
Σκοπός της πτυχιακής αυτής, είναι η σχεδίαση και ανάπτυξη συστήματος προσομοίωσης χρονοδρομολόγησης με τυχαίες τιμές παραμέτρων (χρόνος εισαγωγής, χρόνος εκτέλεσης, προτεραιότητα) και πιθανότητας λειτουργίας Down σε σημαφόρο του συστήματος. Η ανάπτυξη του κώδικα έχει γίνει σε δύο «σκέλη». Στο πρώτο υλοποιήθηκαν όλες οι μέθοδοι χρονοπρογραμματισμού και υπολογίστηκαν οι χρόνοι:

1. Αναμονής
2. Εκτέλεσης
3. Ολοκλήρωσης

Και στο δεύτερο σκέλος, υλοποιήθηκε η μέθοδος χρονοπρογραμματισμούπου ισχύει στα σύγχρονα λειτουργικά συστήματα.

2.1 Πρόγραμμα παραγωγής αρχείων εισόδου

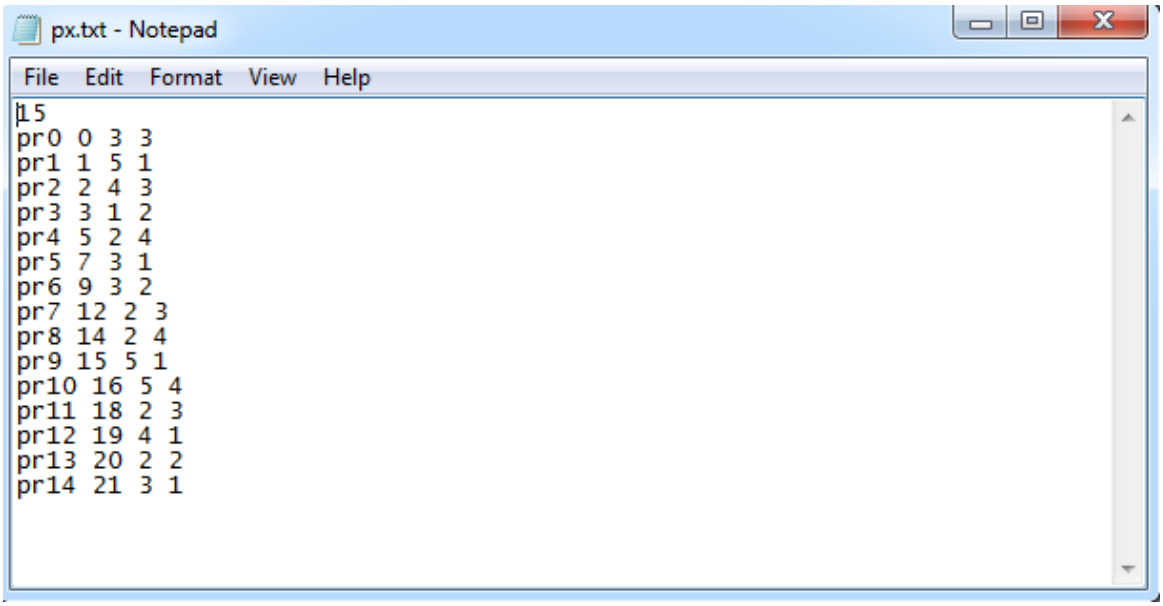
Πρώτα από όλα δημιουργήθηκε ένα βοηθητικό πρόγραμμα παραγωγής αρχείων εισόδου επιθυμητού μεγέθους αναλόγων των αναγκών μας. Για διευκόλυνση έχει σταθερή μορφή. Το πρόγραμμα ονομάζεται «create_process.c» ζητά από το χρήστη το πλήθος των εγγραφών που θέλει να περιέχονται, την πιθανότητα εμφάνισης της κάθε διεργασίας, το μέγιστο χρόνο εκτέλεσηςκαι τη μέγιστη προτεραιότητα, όπως φαίνεται στην εικόνα.



Εικόνα 2.1 Οθόνη δημιουργίας αρχείου εισαγωγής

Στο συγκεκριμένο παράδειγμα έχει ονομαστεί το αρχείο: px, είναι της μορφής .txt και περιέχει 15 διεργασίες με 80% πιθανότητα εμφάνισης την κάθε μιας, με μέγιστο αριθμό

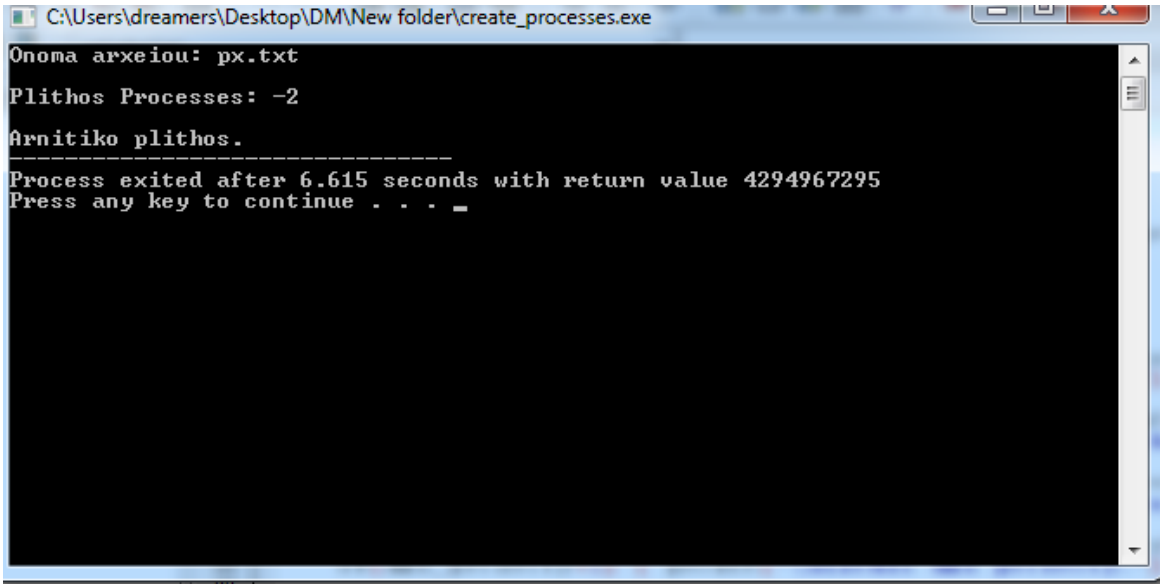
αιτημάτων προς τη CPU ίσο με 5 και προτεραιότητα το μέγιστο. Το αρχείο που εξάγει το πρόγραμμα είναι:



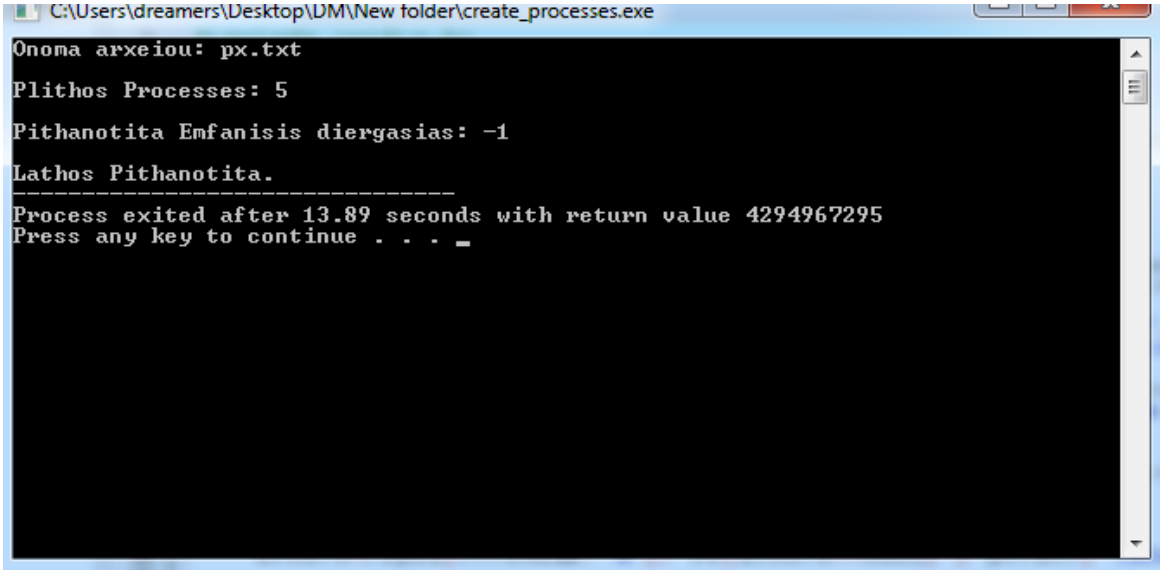
Εικόνα 2.2 Αρχείο εισαγωγής

Η ονομασία της κάθε διεργασίας γίνεται αυτόματα με βήμα 1.

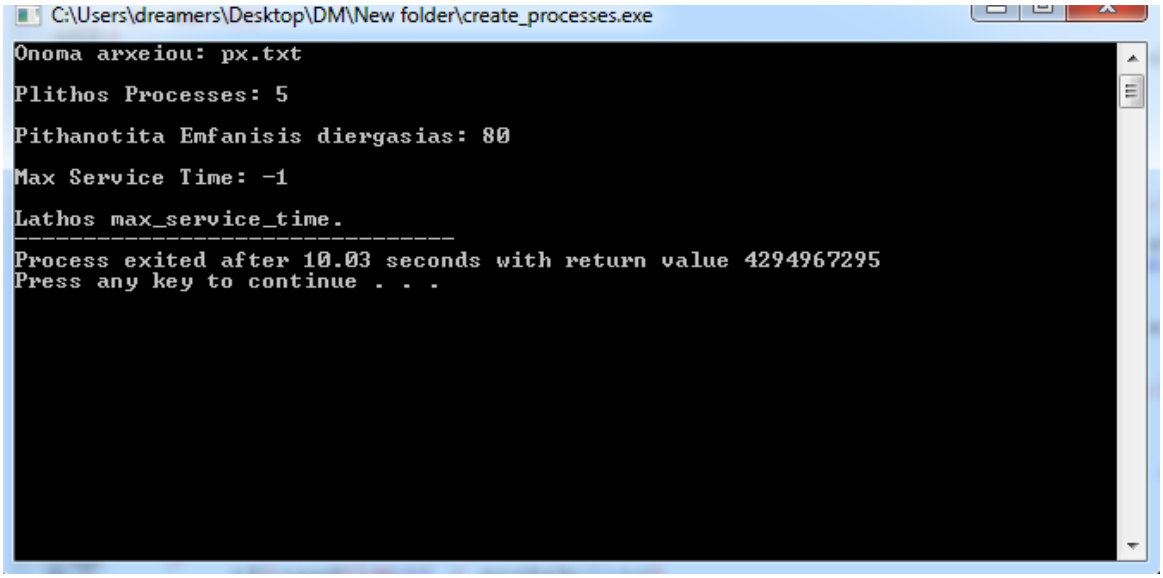
Το πρόγραμμα εκτελεί όλους τους απαραίτητους ελέγχους σχετικά με την είσοδο που παίρνει από το χρήστη και εξάγει τα αντίστοιχα μηνύματα όπως:



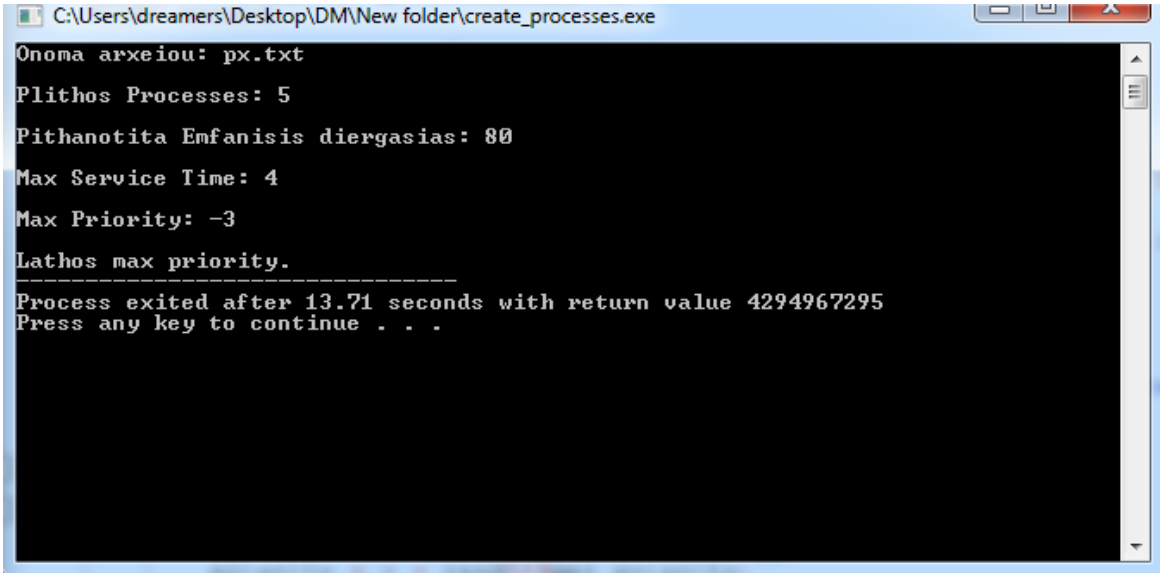
Εικόνα 2.3 Εκτέλεση αρχείου – σφάλμα στο πλήθος των διεργασιών



Εικόνα 2.4 Εκτέλεση αρχείου – σφάλμα στη πιθανότητα εμφάνισης διεργασίας



Εικόνα 2.5 Εκτέλεση αρχείου – σφάλμα στη τιμή του μέγιστου χρόνου εκτέλεσης κάθε διεργασίας



Εικόνα 2.6 Εκτέλεση αρχείου – σφάλμα στη μέγιστη τιμή της προτεραιότητας

Με αυτό τον τρόπο δημιουργήθηκαν τα αρχεία εισόδου για την υλοποίηση και τη δημιουργία αποτελεσμάτων από τον κώδικα.

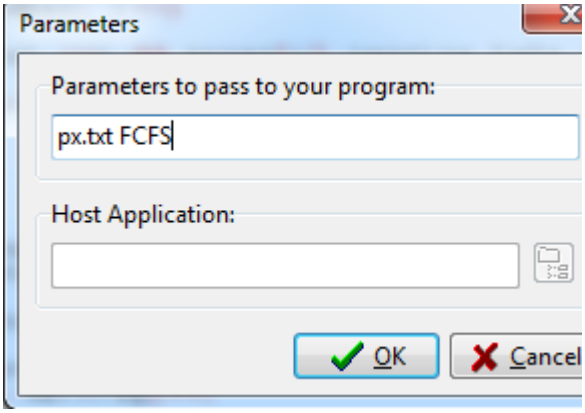
2.2 Μέθοδοι Χρονοπρογραμματισμού

Στο πρώτο σκέλος όπως αναφέρεται παραπάνω, προσομοιάστηκαν οι βασικές πολιτικές χρονοπρογραμματισμού.

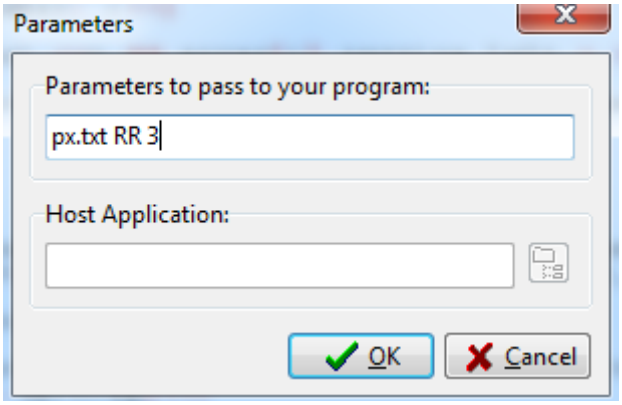
Έτσι δημιουργήθηκε το Project1 που περιέχει τα αρχεία:

- main.c
- function.h
- function.c.

Η main.c παίρνει ως είσοδο το αρχείο εισόδου, την πολιτική που θέλουμε να εκτελέσει και αν είναι η Round Robin και το κβάντο (q).



Εικόνα 2.7 Παράμετροι εισόδου

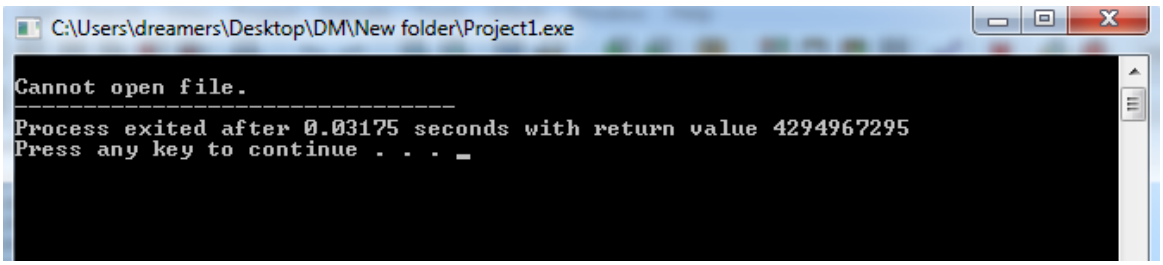


Εικόνα 2.8 Παράμετροι εισόδου με κβάντο

Κάνει τους απαραίτητους ελέγχους αν τα στοιχεία εισόδου – ορίσματα (arguments) είναι σωστά και αντιστοιχεί με την αντίστοιχη μέθοδο χρονοπρογραμματισμού. Αν είναι λάθος εμφανίζονται τα κατάλληλα μηνύματα σφάλματος έπειτα από τους ελέγχους:

a. Λάθος αρχείο εισόδου

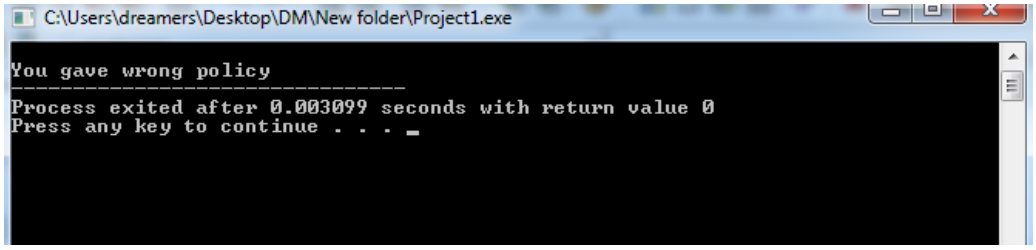
```
arxeio=fopen(argv[1],"r");
if(arxeio==NULL)
{ printf("\nCannot open file."); exit(-1); }
```



Εικόνα 2.9 Αδύνατη ανάγνωση του αρχείου

b. Λάθος πολιτική

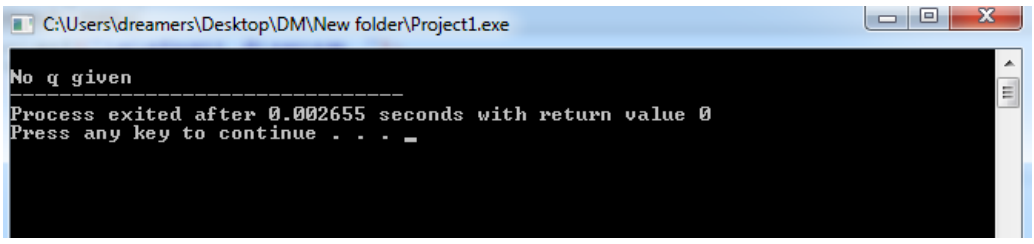
```
if(strcmp(argv[2], "FCFS") == 0) { policy = 1; }
else if(strcmp(argv[2], "SJFnp") == 0) { policy = 2; }
else if(strcmp(argv[2], "SJFp") == 0) { policy = 3; }
else if(strcmp(argv[2], "PRnp") == 0) {policy = 4; }
else if(strcmp(argv[2], "PRp") == 0) { policy = 5; }
else if(strcmp(argv[2], "RR") == 0)
{     policy = 6;
      if(argc==3) { printf("\nNo q given"); exit(0);  }
      q = atoi(argv[3]);
      if(q<=0) { printf("\nq<=0"); exit(0);}
}
else { printf("\nYou gave wrong policy"); exit(0); }
```



Εικόνα 2.10 Μήνυμα λάθους πολιτικής

c. Παράλειψη κβάντου

```
if(strcmp(argv[2], "RR") == 0)
{
      policy = 6;
      if(argc==3)
      {
            printf("\nNo q given");
            exit(0);
      }
      q = atoi(argv[3]);
      if(q<=0)
      {
            printf("\nq<=0");
            exit(0);
      }
}
```



Εικόνα 2.11 Μήνυμα έλλειψης κβάντου

Αφού ολοκληρωθούν οι παραπάνω έλεγχοι , διαβάζει το αρχείο και δεσμεύει την απαραίτητη μνήμη και δημιουργεί πίνακα με τις εγγραφές του αρχείου. Εκτυπώνει τις εγγραφές που διάβασε από το αρχείο εισόδου και έπειτα καλεί την πολιτική μέσω συναρτήσεων.

Κάθε εγγραφή είναι της μορφής:

```
typedef struct
{
      char name[8];
      int arrival;
      int service;
      int priority;
      int service_left;
      int waiting;
      intends;
      int state;
}process;
```

Και περιέχει ένα πίνακα χαρακτήρων για το όνομα κάθε διεργασίας, επτά (7) μεταβλητές integer: για το χρόνο εκτέλεσης, χρόνο εισαγωγής, τιμή προτεραιότητας, το χρόνο αναμονής, το χρόνο που τερμάτισε η διεργασία και άλλος ένας integerπου κρατείτε η κατάσταση στην οποία βρίσκεται η διεργασία:

0: δεν έχει έρθει ακόμα η διεργασία

1: σε κατάσταση «running»

2: σε κατάσταση «ready»

3: σε κατάσταση «blocked»

4: σε κατάσταση «terminated»

Η κάθε συνάρτηση που καλείται από το κεντρικό πρόγραμμα ικανοποιεί μία μέθοδο χρονοπρογραμματισμού:

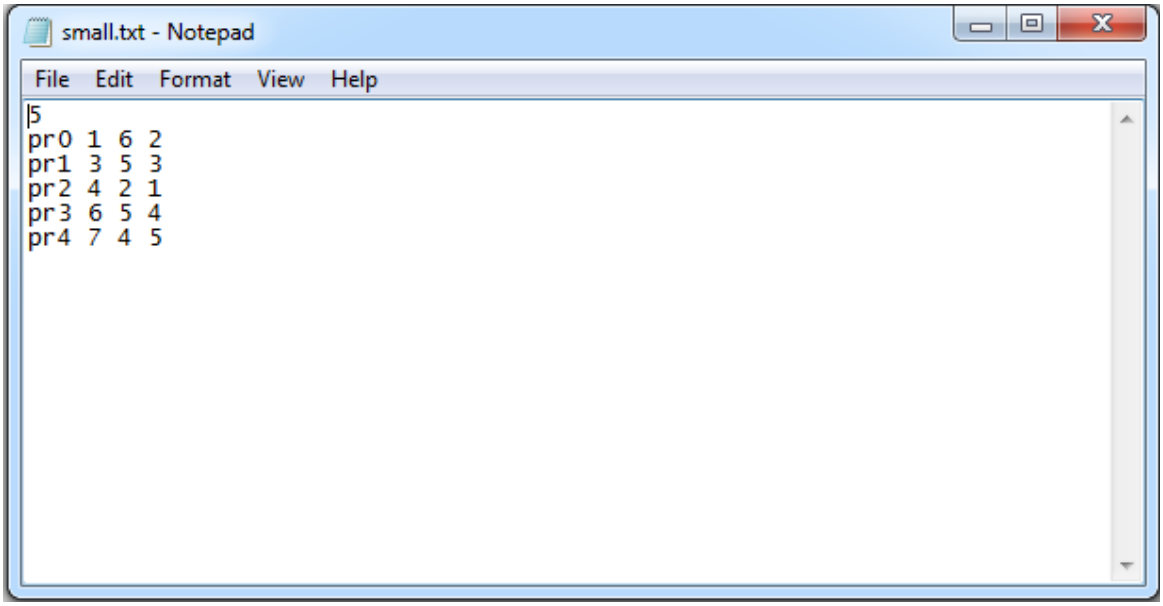
Συνάρτηση First Come First Served

Η συνάρτηση με αυτό το όνομα υλοποιεί τη μέθοδο χρονοπρογραμματισμού «First Come First Served», δέχεται ως είσοδο δείκτη στο πίνακα και το πλήθος των διεργασιών. Εμφανίζει στο χρήστη το διάγραμμα Gantt. Ως κριτήριο είναι ο χρόνος, για κάθετελέγχονται ποιες διεργασίες είναι ενεργές και εκτελείται πάντα αυτή με το μικρότερο χρόνο εισαγωγής «arrival time» μέχρι να εκτελεστεί ολόκληρη. Η κλήση από το κεντρικό πρόγραμμα γίνεται ως εξής:

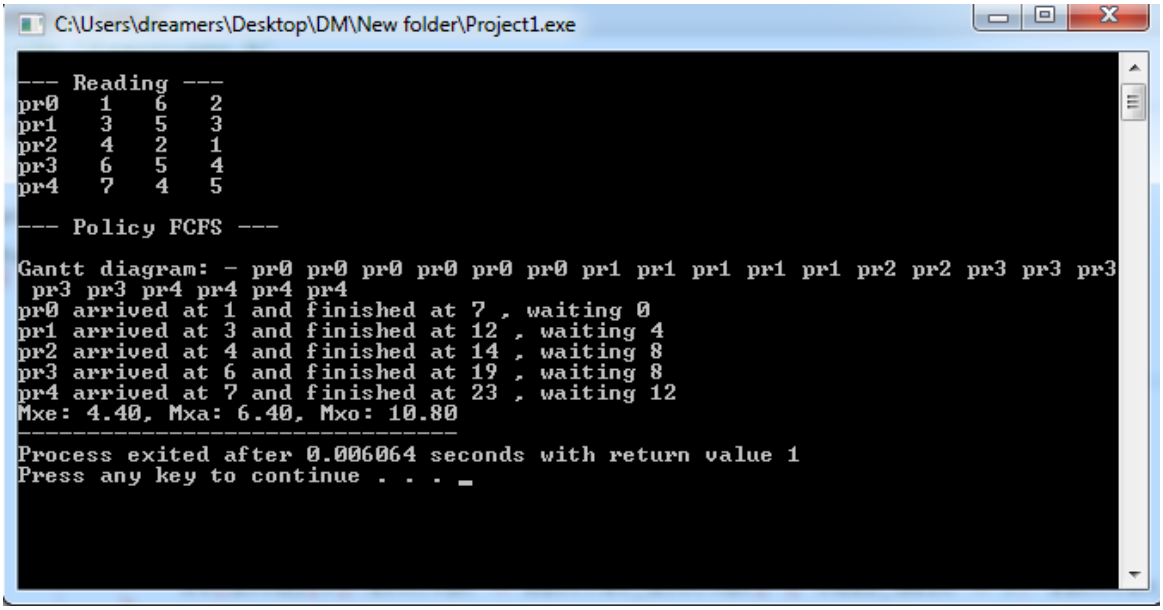
```
if(policy == 1)
{
      FCFS(array, processes_count);
}
```

Ο αλγόριθμος αυτός είναι ίσως ο πιο εύκολος αλγόριθμος χρονοπρογραμματισμού, ανήκει στο μη προεκχωρητικό χρονοπρογραμματισμό. Με τον αλγόριθμο αυτό όποια διεργασία ζητήσει πρώτη την CPU, θα την δεσμεύσει. Η υλοποίηση του αλγορίθμου στηρίζεται στη λογική της ουράς FIFO. Όταν μια διεργασία εισέλθει στη λίστα ετοιμότητας, το PCB της συνδέεται στο τέλος της ουράς (PCB = η λίστα από εγγραφές που είναι δείκτες σε μπλοκ ελέγχου διεργασιών). Όταν ο επεξεργαστής είναι ελεύθερος, τον δεσμεύει η διεργασία που βρίσκεται στην κορυφή της ουράς και αφαιρείται από την ουρά. Παρόλο που ο αλγόριθμος FCFS είναι εύκολος για να υλοποιηθεί αγνοεί όλα τα κριτήρια τα οποία επηρεάζουν την επίδοση. Το μειονέκτημα αυτού του αλγορίθμου είναι ο σχετικά μεγάλος χρόνος αναμονής, καθώς επίσης, και τα προβλήματα που δημιουργεί σε συστήματα time – sharing, όπου ο κάθε χρήστης χρειάζεται το μερίδιο του από τον επεξεργαστή ανά τακτά χρονικά διαστήματα. Θα είναι καταστροφικό να επιτρέψουμε σε μία διεργασία να κρατήσει δεσμευμένο τον επεξεργαστή για μεγάλο χρονικό διάστημα.

Το αρχείο εισόδου και το αποτέλεσμα που εμφανίζεται στην οθόνη:



Εικόνα 2.11 Αρχείο Εισόδου



Εικόνα 2.12 Αποτέλεσμα πολιτικής FCFS

Συνάρτηση Shortest Job First

Ο συγκεκριμένος αλγόριθμος αφορά εργασίες δέσμης (batch jobs), των οποίων οι χρόνοι εκτέλεσης είναι γνωστοί εκ των προτέρων. Κριτήριο του αλγορίθμου αυτού, είναι η επιλογή διεργασίας με το μικρότερο χρόνο εκτέλεσης (service time). Ο αλγόριθμος αυτός ελαχιστοποιεί το μέσο χρόνο αναμονής γιατί εξυπηρετούνται οι μικρές διεργασίες πριν από τις μεγάλες διεργασίες. Ενώ όμως ελαχιστοποιεί το μέσο χρόνο αναμονής, καταδικάζει τις διεργασίες με μεγάλο χρόνο εκτέλεσης. Για παράδειγμα, αν η λίστα ετοιμότητας είναι κατακερματισμένη, τότε οι διεργασίες τείνουν να ξεχνιούνται στη λίστα ενώ οι μικρότερες διεργασίες εκτελούνται. Στην εξαιρετική περίπτωση που το σύστημα έχει λίγους χρόνους αδράνειας (idle time), οι μεγάλες διεργασίες δεν πρόκειται να εκτελεστούν ποτέ. Ο αλγόριθμος αυτός ανήκει και στους προεκτοπιστικούς και στους μη.

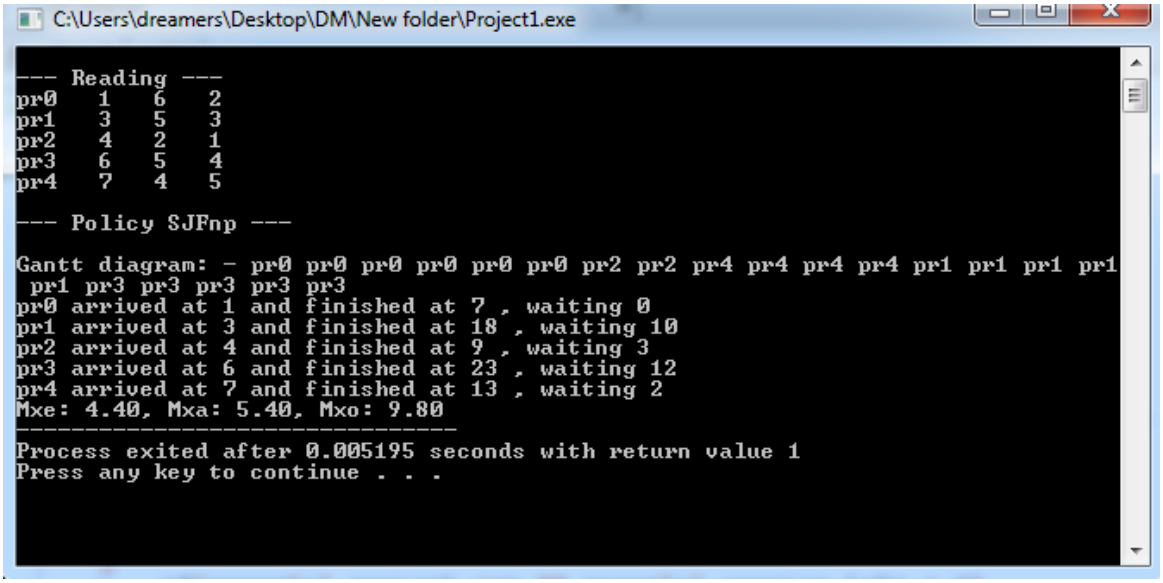
Συνάρτηση SJFnp

Ησυνάρτηση με αυτό το όνομα υλοποιεί τη μέθοδο χρονοπρογραμματισμού «Shortest Job First non-preemptive», δέχεται ως είσοδο δείκτη στο πίνακα διεργασιών και το πλήθος των διεργασιών και εμφανίζει στο χρήστη το διάγραμμα Gantt. Ως κριτήριο είναι ο χρόνος για κάθε τελέγχεται ποιες διεργασίες είναι ενεργές και εκτελείται πάντα αυτή με το μικρότερο χρόνο εκτέλεσης «service time». Από τη στιγμή που επιλέγεται μία διεργασία δεν μπορεί να τη διακόψει κάποια άλλη μέχρι να εκτελεστεί ολόκληρη.

Η συνάρτηση καλείται από τη main.c:

```
if(policy == 2)
{
	SJFnp(array,rocesses_count);
}
```

Το αρχείο εισόδου είναι κοινό για όλες τις συναρτήσεις και το αποτέλεσμα που βλέπουμε στην οθόνη είναι:



Εικόνα 2.13 Αποτέλεσμα πολιτικής SJFnp

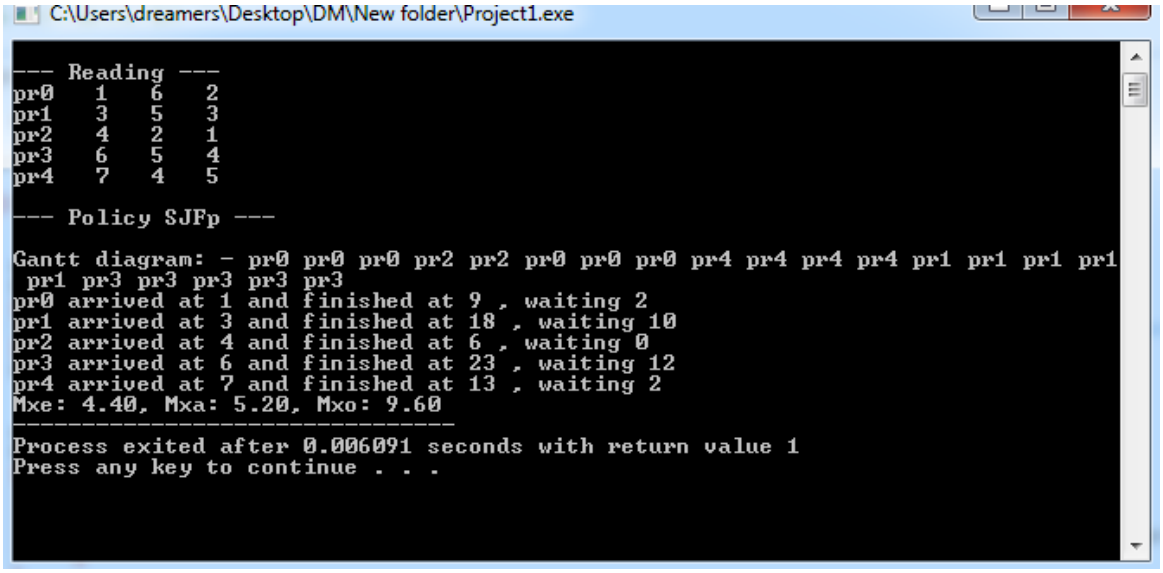
Συνάρτηση SJFp

Η συνάρτηση με αυτό το όνομα υλοποιεί τη μέθοδο χρονοπρογραμματισμού «Shortest Job First preemptive», δέχεται ως είσοδο δείκτη στο πίνακα διεργασιών και το πλήθος των διεργασιών και εμφανίζει στο χρήστη το διάγραμμα Gantt. Ως κριτήριο είναι ο χρόνος για κάθε τελέγχεται ποιες διεργασίες είναι ενεργές και εκτελείται πάντα αυτή με το μικρότερο χρόνο εκτέλεσης «service time». Ο έλεγχος γίνεται κάθε t και επιλέγεται πάντα αυτή που ικανοποιεί το παραπάνω κριτήριο.

Η συνάρτηση καλείται από τη main.c:

```
if(policy == 3)
{
	SJFp(array,processes_count);
}
```

Το αρχείο εισόδου είναι κοινό για όλες τις συναρτήσεις και το αποτέλεσμα που βλέπουμε στην οθόνη είναι:



Εικόνα 2.14 Αποτέλεσμα πολιτικής SJFp

Συνάρτηση Priority Scheduling

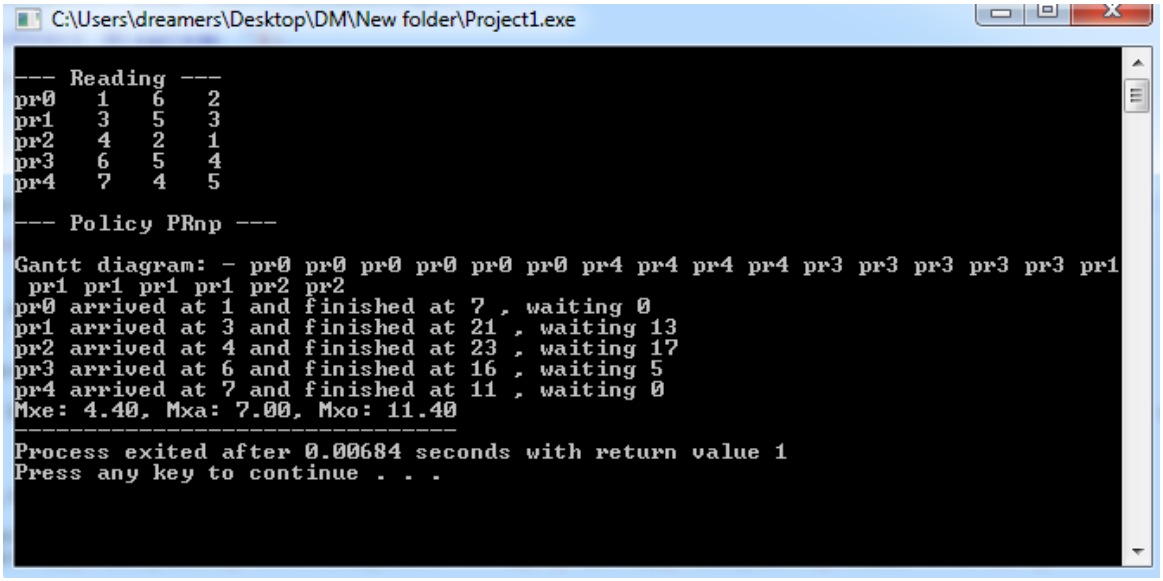
Ο αλγόριθμος αυτός είναι παρόμοιος με τον ShortestJobFirst (SJF) με κριτήριο όχι ο χρόνος εκτέλεσης (service time) αλλά η προτεραιότητα (priority) της κάθε διεργασίας. Και εδώ υπάρχουν δύο κατηγορίες: προεκχωρητικός και μη προεκχωρητικός. Εδώ ο αλγόριθμος δίνει τους πόρους της CPU στη διεργασία που βρίσκεται σε κατάσταση «έτοιμη» (ready) με τη μεγαλύτερη προτεραιότητα (priority).

Συνάρτηση PRnp

Η συνάρτηση με αυτό το όνομα υλοποιεί τη μέθοδο χρονοπρογραμματισμού «Priority Scheduling non-preemptive», δέχεται ως είσοδο δείκτη στο πίνακα διεργασιών και το πλήθος των διεργασιών και εμφανίζει στο χρήστη το διάγραμμα Gantt. Ως κριτήριο είναι ο χρόνος για κάθε τελέγχεται ποιες διεργασίες είναι ενεργές και εκτελείται πάντα αυτή με τη μεγαλύτερη προτεραιότητα «priority». Ο έλεγχος γίνεται κάθε t και από τη στιγμή που έχει επιλεχθεί μια διεργασία δεν διακόπτεται από καμία άλλη.

Με το ίδιο αρχείο εισόδου, η κλήση και το αποτέλεσμα που βλέπουμε στην οθόνη είναι:

```
if(policy == 4)
{
      PRnp(array,processes_count);
}
```



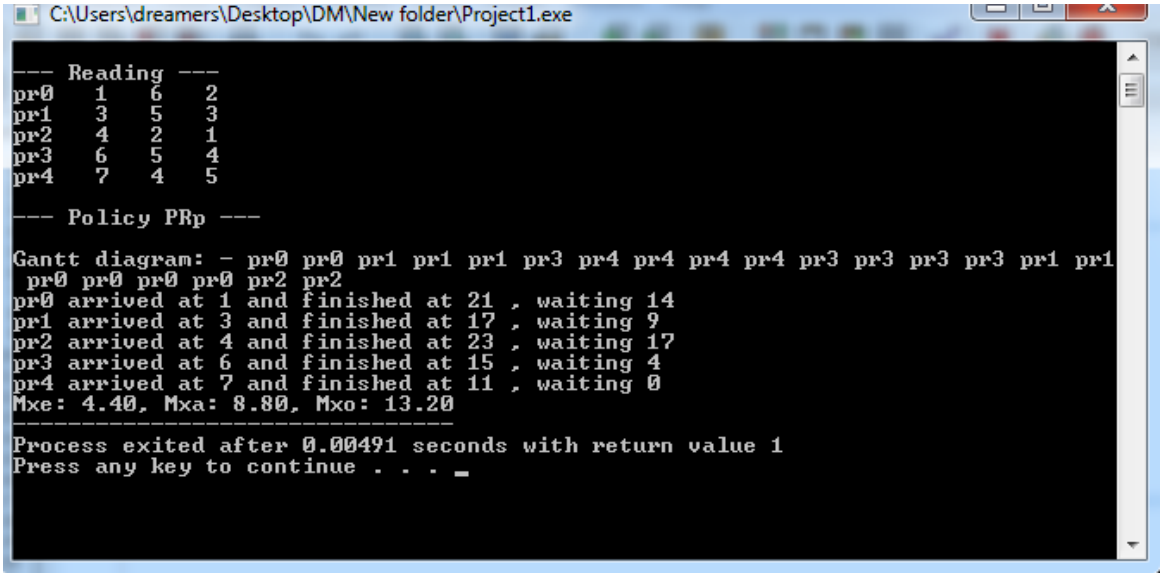
Εικόνα 2.15 Αποτέλεσμα πολιτικής PRnp

Συνάρτηση PRp

Η συνάρτηση με αυτό το όνομα υλοποιεί τη μέθοδο χρονοπρογραμματισμού «Priority Scheduling preemptive», δέχεται ως είσοδο δείκτη στο πίνακα διεργασιών και το πλήθος των διεργασιών και εμφανίζει στο χρήστη το διάγραμμα Gantt. Ως κριτήριο είναι ο χρόνος για κάθε τελέγχεται ποιες διεργασίες είναι ενεργές και εκτελείται πάντα αυτή με τη μεγαλύτερη προτεραιότητα «priority». Ο έλεγχος γίνεται για κάθε t και εκτελείται πάντα αυτή με τη μεγαλύτερη προτεραιότητα.

Η κλήση της συναρτήσεως γίνεται με την παρακάτω εντολή και το αποτέλεσμα που βλέπουμε στην οθόνη, με το ίδιο αρχείο εισόδου, είναι:

```
if(policy == 5)
{
      PRp(array,processes_count);
}
```

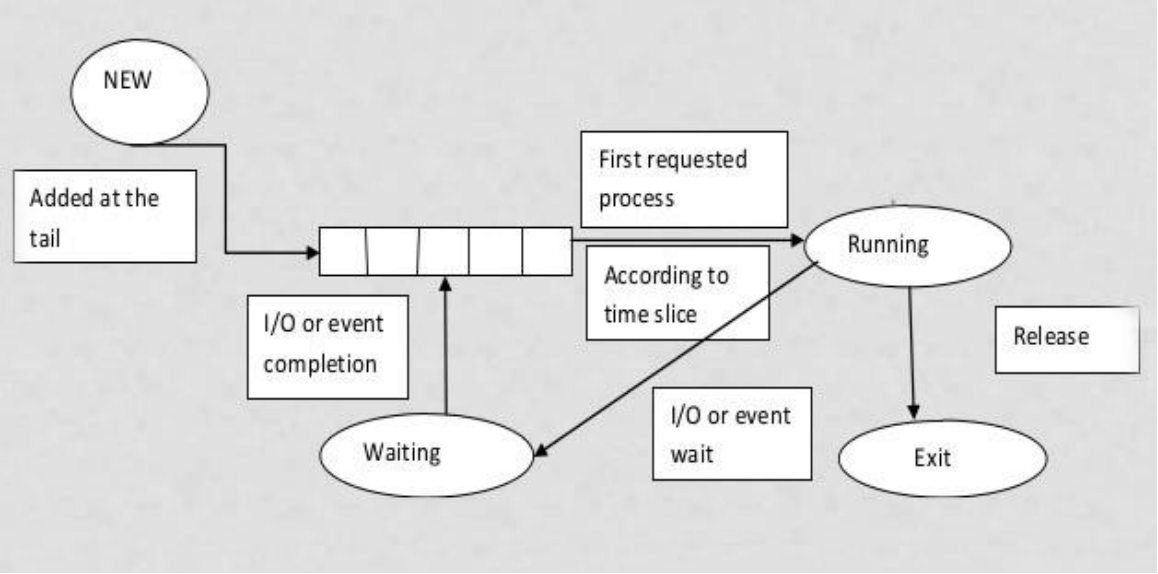


Εικόνα 2.16 Αποτέλεσμα πολιτικής PRp

Συνάρτηση Round Robin

Ο αλγόριθμος χρονοπρογραμματισμού εξυπηρέτησης εκ περιτροπής (RoundRobin - RR) χαρακτηρίζεται ως ο πιο δίκαιος και διαδεδομένος αλγόριθμος εφόσον δεν προκαλεί στέρηση (starvation) στο λειτουργικό σύστημα και στις διεργασίες αυτού. Ο αλγόριθμος αυτός αποτελείται από μία ουρά first-in-first-out (FIFO) για τις διεργασίες που βρίσκονται σε κατάσταση «έτοιμη» (ready) και παραχωρεί δικαίωμα συγκεκριμένου χρόνου σε κάθε διεργασία που βρίσκεται σε κατάσταση «εκτελέσιμη» (running). Το ορισμένο αυτό χρονικό διάστημα εκτέλεσης κάθε διεργασίας ονομάζεται κβάντο (quantum) και συμβολίζεται με «q». Όταν η εκάστοτε διεργασία ολοκληρώσει το χρονικό αυτό διάστημα, μπλοκάρεται (κατάσταση blocked) και επιστρέφει στην ουρά αναμονής.

Ο αλγόριθμος του RR, γενικά, είναι εύκολος στην κατανόησή του και φαίνεται στην παρακάτω εικόνα.



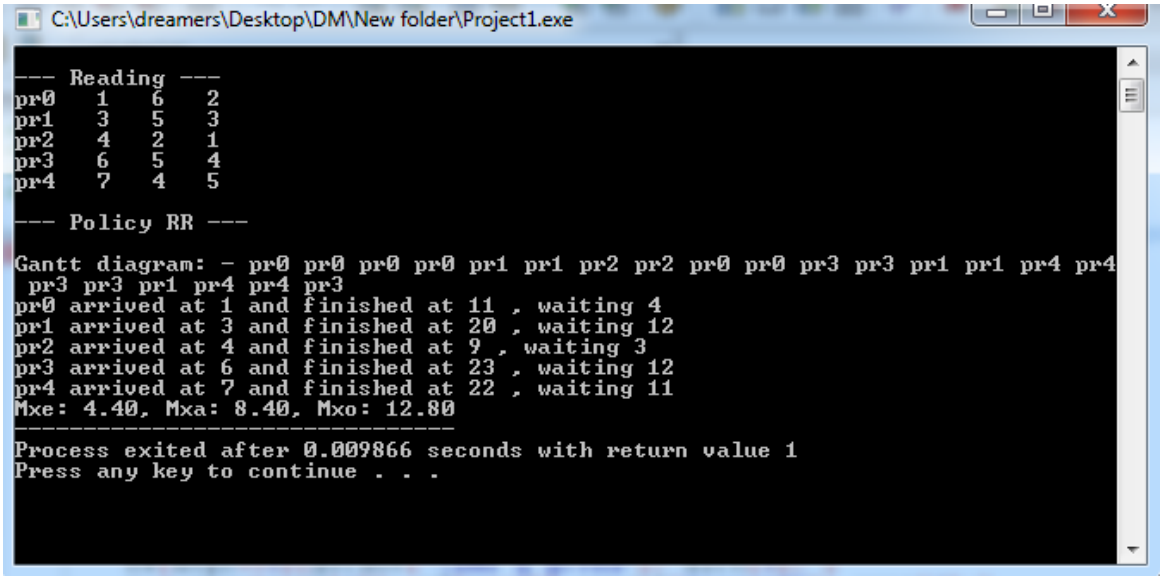
Εικόνα 2.17 Πλήρες σχήμα Εκ Περιτροπής (Round Robin)

Το μόνο που έχει να κάνει ο χρονοδρομολογητής είναι να διατηρήσει μία λίστα με τις εκτελούμενες διεργασίες. Όταν μια διεργασία χρησιμοποιήσει το κβάντο της, μπαίνει στο τέλος της λίστας.

Η συνάρτηση αυτή, δέχεται ως είσοδο δείκτη στο πίνακα διεργασιών, το πλήθος των διεργασιών και το κβάντο «q», έπειτα εμφανίζει στο χρήστη το διάγραμμαGantt. Κάθε διεργασία που αιτείται τη CPU εισέρχεται σε μία ουρά λογικής FIFO. Ο έλεγχος γίνεται για κάθε χρονική στιγμή t. Όταν μια διεργασία της δοθεί δικαίωμα εκτέλεσης, το οποίο το διατηρεί γιαχρονική περίοδο όσο έχει ορισθεί το κβάντο κατά την κλήση της συνάρτησης. Όλες οι υπόλοιπες διεργασίες που ενεργοποιούνται «στοιβάζονται» στην ουρά FIFO.Αφούη εκτελέσιμη διεργασία ολοκληρώσει το χρόνο εκτέλεσής της, τότε ελευθερώνει τη CPU και αν έχει υπολειπόμενο χρόνο εκτέλεσης πάει στο τέλος της ουράς. Η πρώτη διεργασία αρχίζει να εκτελείται. Η κλήση συνάρτησης γίνεται από το κεντρικό πρόγραμμα με την εντολή:

```
if(policy == 6)
{
      RR(array,processes_count,q);
}
```

Τα αποτελέσματα είναι τα παρακάτω αν ορίσουμε κβάντο q = 2:



Εικόνα 2.18 Αποτέλεσμα πολιτικής R-R

Υπολογισμός χρόνων εκτέλεσης

Εκτός από την υλοποίηση των μεθόδων κρατήθηκαν στατιστικά με την κλήση της συνάρτησης:

```
print_stats(array, processes_count);
```

Η συνάρτηση αυτή υπολογίζει τρείς χρόνους:

1. M.X.E (μέσος χρόνος εξυπηρέτησης)

```
for(i=0; i<proccess_number; i++)
{
      mxe += (array[i].service);
}
mxe = mxe/(float)proccess_number;
```

Ο μέσος χρόνος εξυπηρέτησης είναι το άθροισμα του χρόνου εκτέλεσης προς το πλήθος των διεργασιών.

2. M.X.A (μέσος χρόνος αναμονής)

```
for(i=0; i<proccess_number; i++)
{
      mxa += (array[i].waiting);
}
mxa = mxa/(float)proccess_number;
```

Κατά την δημιουργία κάθε διεργασίας στο πίνακα διεργασιών δημιουργείται ένα πεδίο integerμε το όνομα«waiting», όπου αρχικοποιείται με την τιμή 0 και κάθε φορά που μία ενεργή διεργασία δεν έτρεχε η διεργασία αυτή αυξάνεται κατά 1. Έτσι για τον υπολογισμό του μέσου χρόνου αναμονής προστίθενται αυτοί οι χρόνοι και διαιρούνται με το πλήθος των διεργασιών.

3. M.X.O (μέσος χρόνος ολοκλήρωσης)

```
for(i=0; i<proccess_number; i++)
{
      mxo += (array[i].ends - array[i].arrival);
}
mxo = mxo/(float)proccess_number;
```

Ο χρόνος ολοκλήρωσης για κάθε διεργασία είναι η διαφορά του χρόνου που εισαγωγής από το χρόνο ολοκλήρωσης της. Και ο μέσος χρόνος είναι το αποτέλεσμα του αθροίσματος όλων των διαφορών προς το πλήθος των διεργασιών.

Σε αυτό το σημείο εύλογο θα ήταν η σύγκριση των αποτελεσμάτων που παρατηρείται στο απλό αυτό παράδειγμα που φαίνεται κάτω από το κάθε συνάρτηση:

Αρχείο εισόδου:

Process_name	Arrival_time	Max_service_time	Priority
pr0	1	6	2
pr1	3	5	3
pr2	4	2	1
pr3	6	5	4
Pr4	7	4	5

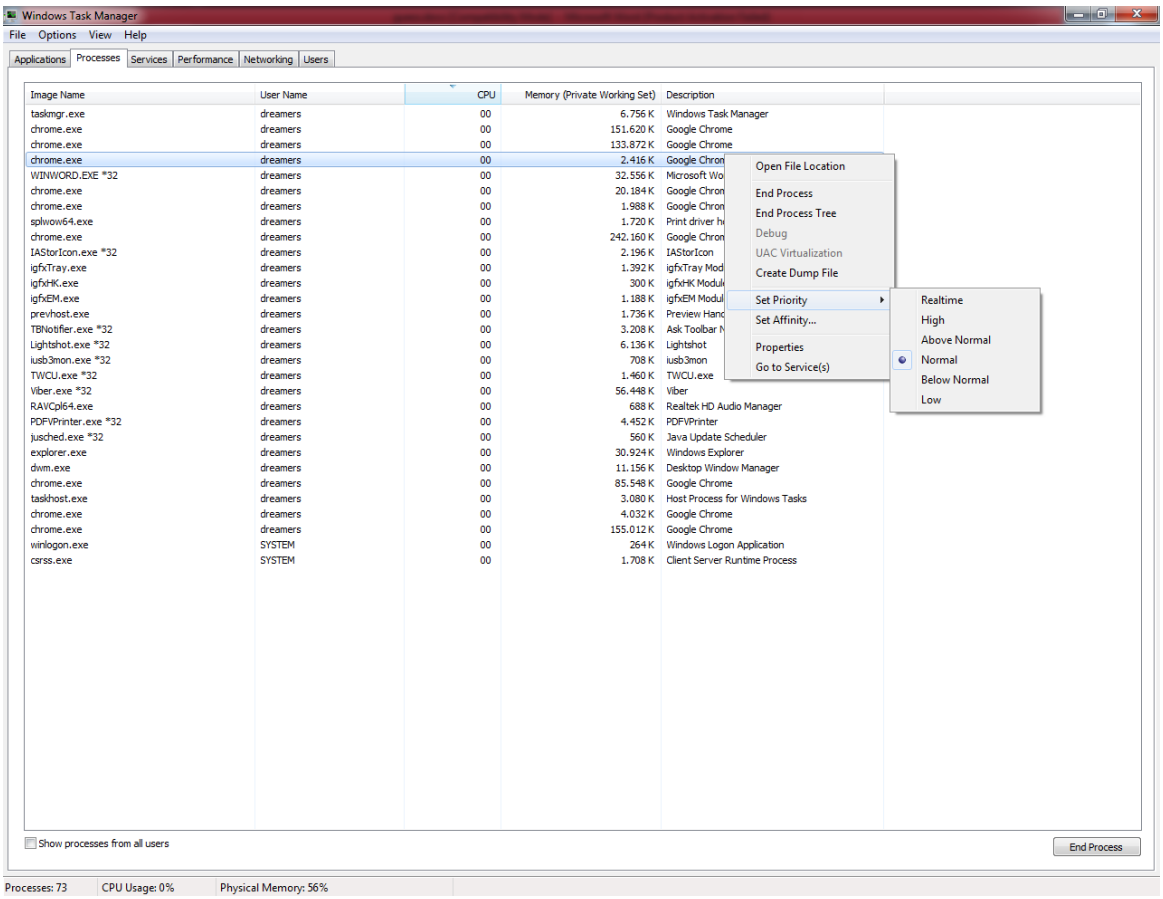
Αποτελέσματα:

Policy	M.X.O	M.X.A	M.X.E
FCFS	10.80	6.40	4.40
SJFnp	9.80	5.40	4.40
SJFp	9.60	5.20	4.40
PRnp	11.40	7.00	4.40
PRp	13.20	8.80	4.40
RR (q=2)	12.80	8.40	4.40

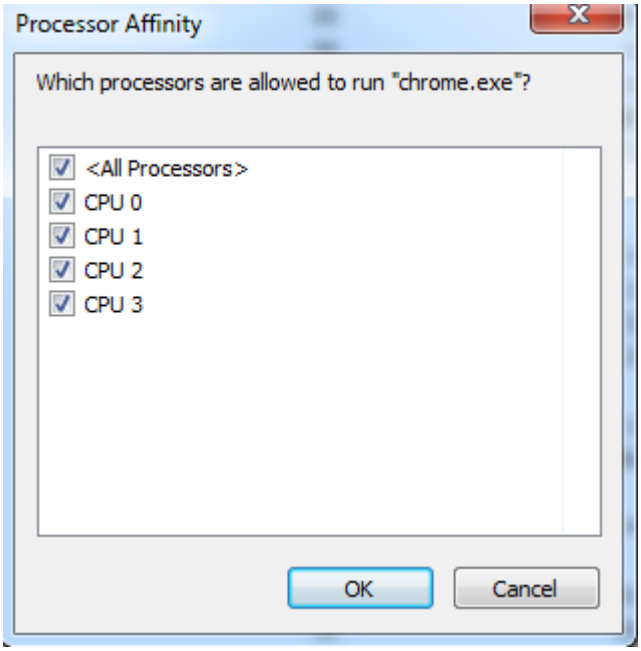
Κεφάλαιο 3 - Χρονοπρογραμματισμός με Εξαρτήσεις Διεργασιών (Σημαφόροι)

3.1 Σύγχρονα λειτουργικά συστήματα

Αφού υλοποιήθηκαν όλες οι βασικές μέθοδοι χρονοπρογραμματισμού θα ήταν αδιανόητο να μην μελετηθεί τι ισχύει στα σύγχρονα λειτουργικά συστήματα. Σήμερα οι υπολογιστές χρησιμοποιούν τη μέθοδο Priority Scheduling προεκτοπιστική.



Εικόνα 3.1 Καρτέλα Ενεργών Διεργασιών Λειτουργικού Συστήματος



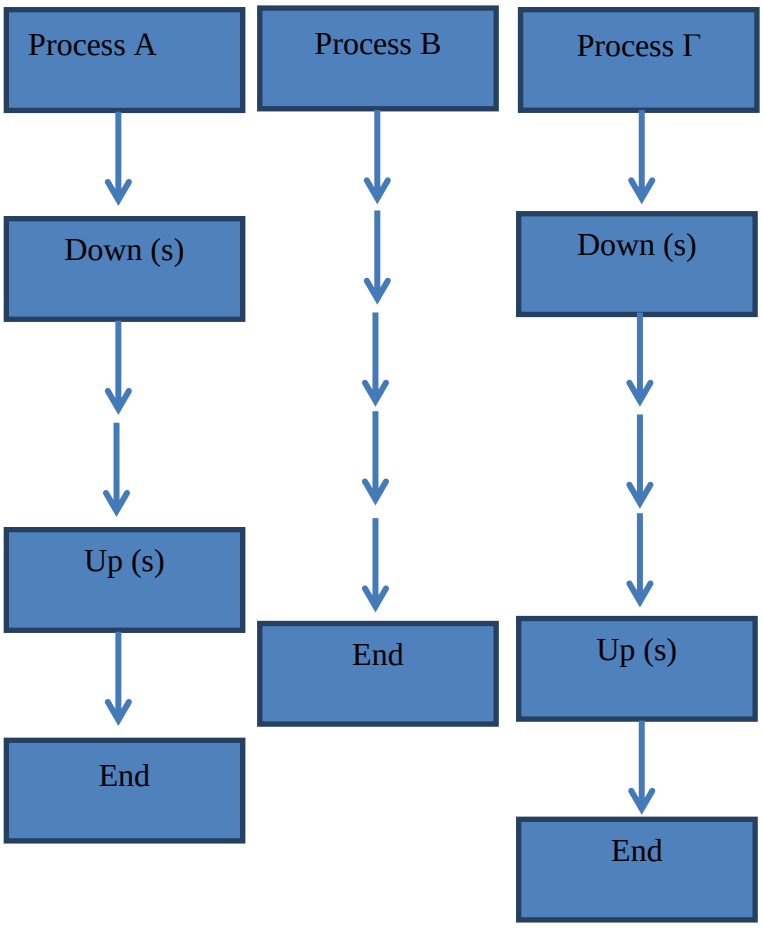
Εικόνα 3.2 Επιλογή Επεξεργαστών προς Εκτέλεση

Στα σύγχρονα, λοιπόν, λειτουργικά συστήματα υπάρχει η δυνατότητα ορισμού προτεραιοτήτων και πόσων επεξεργαστών (CPU) θα χρησιμοποιηθούν. Έτσι υλοποιήθηκε η μέθοδος αυτή και ελέγχθηκαν οι εξαρτήσεις που υπάρχουν. Όταν αναφέρεται ο όρος «εξαρτήσεις» εννοείται η προσομοίωση των σημαφόρων. Η καλύτερη κατανόηση των σημαφόρων και του σημαντικού ρόλου που έχουν στο λειτουργικό σύστημα, γίνεται μέσα από την μελέτη του εξής προβλήματος:

Αν θεωρηθεί ότι υπάρχουν τρεις διεργασίες Α, Β, Γ με μειούμενες προτεραιότητες και τη διεργασία Α να φέρει την υψηλότερη(priorityA > priorityB > priorityΓ). Οι διεργασίες Α και Γ χρησιμοποιούν έναν κοινό πόρο. Ο πόρος αυτός προστατεύεται από έναν σημαφόρο για την αποφυγή ταυτόχρονης πρόσβασης των δύο αυτών διεργασιών. Το παρακάτω σχήμα δείχνει τη δομή Α, Β, Γ. Οι χρόνοι συμβολίζονται με βέλη, κάθε βέλος είναι ίσο με Τ. Οι διεργασίες εκτελούνται (μόνο μία φορά) στους ακόλουθους χρόνους (arrivaltime):

- Α = 2Τ
- Β = 4Τ
- Γ = 0Τ.

Θεωρείται ότι η εφαρμογή προεκτόπισης χρονοδρομολόγησης είναι με σταθερές προτεραιότητες. Η αρχική τιμή του σημαφόρου είναι 1.



Εικόνα 3.3 Παράδειγμα Εκτέλεσης Διεργασιών, με χρήση Σημαφόρων

Στην εικόνα φαίνονται οι διεργασίες και ο σημαφόρος. Θα μελετηθεί με κριτήριο το χρόνο εκτέλεσης.

Για t=0 T: η διεργασία Γ θα αιτηθεί τη CPU και από την κατάσταση «ready» θα μεταβεί στη κατάσταση «Running», αφού η CPU είναι ελεύθερη.

Για t=1T:θα γίνει ο μετρητής του σημαφόρου 0 (ήταν 1 – 1=0) και θα συνεχίσει να εκτελείται η διεργασία Γ.

Για t=2T: θα αιτηθεί τη CPU και η διεργασία Α, λόγο της μεγαλύτερης προτεραιότητας (priorities: A>B>Γ) θα αλλάξει η κατάσταση της διεργασίας Γ σε«waiting» και η κατάσταση της Α σε «running»

Για t=3T: η διεργασία Α θα αιτηθεί να κάνει Downστο σημαφόρο (η κατάσταση του σημαφόρου s=0) δηλαδή 0 –1 =–1 , αυτό το αίτημα δεν μπορεί να πραγματοποιηθεί άρα η διεργασία Α μπλοκάρεται (κατάσταση Α «blocked») και τρέχει η διεργασία Γ που είναι η μόνη ενεργή (κατάσταση Γ «running»).

Για t=4T:ενεργοποιείται και η διεργασία Β η οποία έχει προτεραιότητα μεγαλύτερη της Γ άρα η κατάσταση της Γ γίνεται «waiting» και η κατάσταση της Β «running».

Η Β θα εκτελεστεί ολόκληρη και την στιγμή t=9T θα τερματίσει.

Για t= 10T: η διεργασία Γ θα αρχίσει να εκτελείται ξανά, τότε θα εκτελεστεί η εντολή Up για το σημαφόρο (0+1=1) και θα μπορέσει η διεργασία Α να εκτελέσει επιτυχημένα το Down του σημαφόρου (1 – 1=0). Τότε παρατηρούμε την εναλλαγή των καταστάσεων των διεργασιών:

Η διεργασία Γ από «waiting» σε «running» και πάλι «waiting»

Η διεργασία Α από «blocked» σε «running» και διατήρηση του σημαφόρου στην τιμή 0 (s=0).

Για t = 11T και t=12T εκτελείται η Α λόγο προτεραιότητας και τη στιγμή t=13T εκτελείται το Up του σημαφόρου επιτυχημένα δίνοντάς του την τιμή 1 (s=1).

Για t = 14T τερματίζει η Α και η Γ με τη σειρά της από «waiting» μετατρέπεται σε «running» και τερματίζει t = 15T.

Εύλογο είναι το ερώτημα τι θα γινόταν αν η διεργασία Β που δεν επηρεάζει και δεν επηρεάζεται από το σημαφόρο είχε άπειρο (πολύ μεγάλο) αριθμό βημάτων εκτέλεσης, δηλαδή η ανάγκη της σε χρόνο CPU ήταν άπειρη;

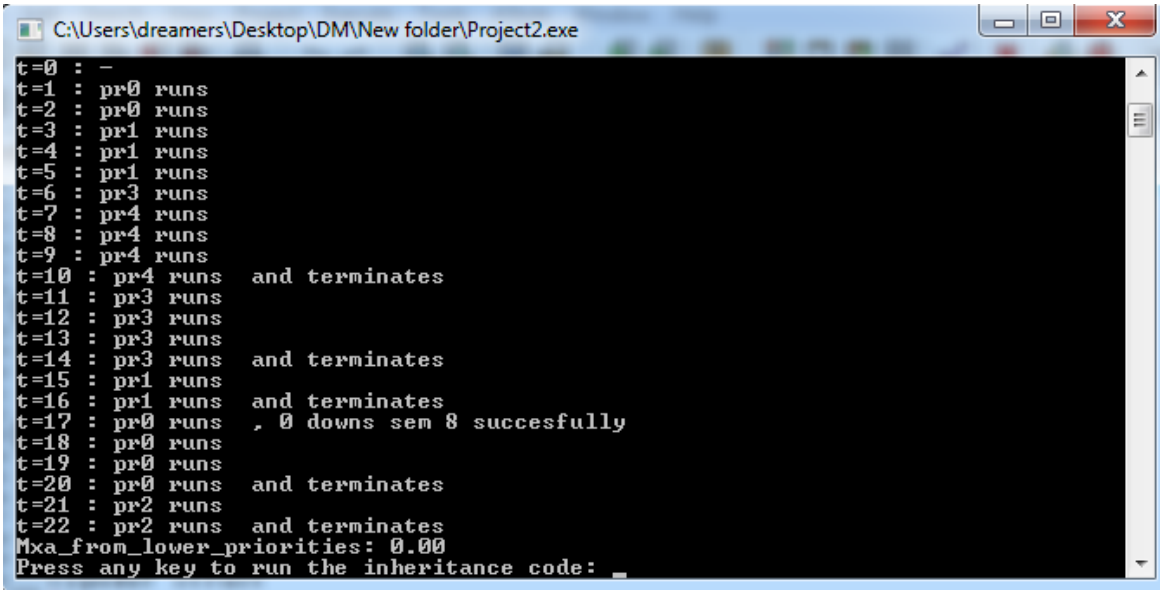
Μέσω της μεθόδου Priority Scheduling Preemptive που υλοποιήθηκε στοπαραπάνωπρόβλημα και ως λύση στο ερώτημα είναι η κληρονομικότητα (inheritance) της προτεραιότητας. Οι σημαφόροι είναι υπεύθυνοι για την αποφυγή συγχρονισμού παράλληλων διεργασιών. Στο παράδειγμα, αν είχε τεθεί αγνώστου – άπειρου χρόνου Τη διεργασία Β, το αποτέλεσμα θα ήταν η διεργασία Α να παραμένει μπλοκαρισμένη από τη Γ για όσο χρονικό διάστημα εκτελείται η Β, παρόλο που η διεργασία Α έχει τη μεγαλύτερη προτεραιότητα (priorityA > priorityB > priorityΓ). Στο πρόγραμμά υπάρχουν δύο συναρτήσεις που εκτελούνται. Στη πρώτη συνάρτηση εκτελείται Priority Scheduling με απλή προσομοίωση σημαφόρων, που ενεργοποιούνται μέσω πιθανότητας που επιθυμεί ο χρήστης. Οι σημαφόροι για κάθε χρονική στιγμή θα βρίσκει έναν αριθμό μέχρι το 100 αν είναι μικρότερος ή ίσος από την πιθανότητα που δόθηκε κατά την εκτέλεση του προγράμματος ο σημαφόρος μεταβαίνει σε κατάσταση «down», αν ο σημαφόρος είναι ήδη σε αυτή τη κατάσταση η διεργασία μπλοκάρεται και αποθηκεύεται στην ουρά των μπλοκαρισμένων διεργασιών από το σημαφόρο αυτό. Στη δεύτερη συνάρτηση η τιμή της προτεραιότητας της μπλοκαρισμένης διεργασίας κληρονομείται μέσω της διαδικασίας που θα αναλύσουμε παρακάτω βάση του παραδείγματος. Δηλαδή, όταν η διεργασία Α μπλοκαριστεί από το σημαφόρο της, ενεργοποιείται η διεργασία Β ενώ έτοιμη για εκτέλεση παραμένει και η διεργασία Γ. Η διεργασία Γ που έχει τη μικρότερη προτεραιότητα και είναι ενεργεί κληρονομεί τη προτεραιότητα της μπλοκαρισμένης διεργασίας, άρα τη χρονική στιγμή t = 3T το priorityΓ κληρονομεί τη προτεραιότητα της διεργασίας Α και τη χρονική στιγμή t = 4T που ενεργοποιείται η διεργασία Β , η προτεραιότητα της είναι μικρότερη της Β άρα αναμένει την ολοκλήρωση της διεργασίας Γ. Με αυτό τον τρόπο η Α θα ενεργοποιηθεί και η Β δεν θα εκτελείται επ’ άπειρον.

3.2 Ανάλυση κώδικα

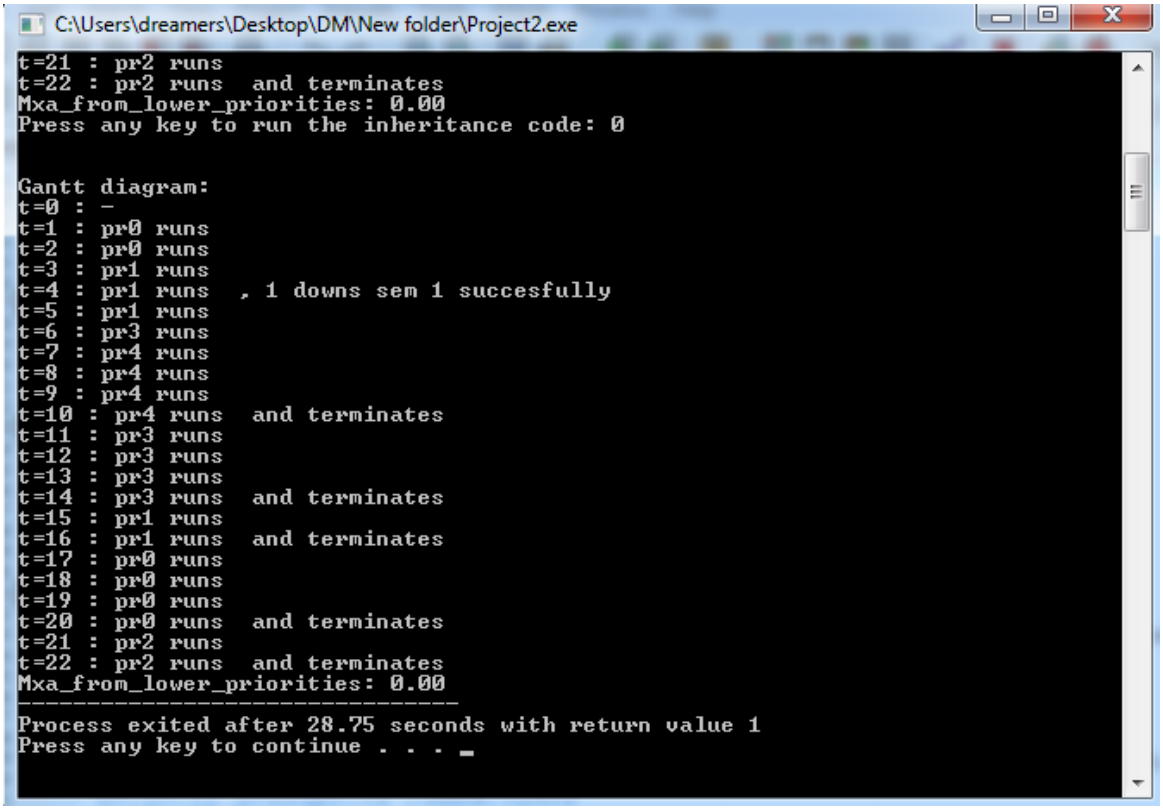
Για την εκτέλεση του προβλήματος ο χρήστης πρέπει να δώσει:

1. Το όνομα του αρχείου
2. Το πλήθος των σημαφόρων
3. Την πιθανότητα που οι σημαφόροι θα μεταβαίνουν στην κατάσταση «Down»
4. Μέσος χρόνος διατήρησης κατάστασης σεμαφόρου

Εκτελώντας το πρόγραμμα το αποτέλεσμα που εμφανίζονται στην οθόνη είναι τα εξής:



Εικόνα 3.4 Αποτέλεσμα εκτέλεσης με πολιτική Priority Scheduling Προεκτοπιστική



Εικόνα 3.5 Αποτέλεσμα εκτέλεσης με πολιτική Priority Scheduling Προεκτοπιστική και Κληρονόμηση Προτεραιοτήτων

Το παράδειγμα αυτό δεν είναι αντιπροσωπευτικό ως προς το χρόνο για έναν υπολογιστή, για αυτό το λόγο δημιουργήθηκαν μεγάλα αρχεία εισόδου με τη βοήθεια του create_processes.c για τον υπολογισμό αντιπροσωπευτικών αποτελεσμάτων και δημιουργία στατιστικών.

Το κεντρικό πρόβλημα και σε αυτή τη περίπτωση θα κάνει τους απαραίτητους ελέγχους:

1. Για τις σωστές παραμέτρους

```
if(argc!=5)
{
      printf("\nWrong arguments.");
      exit(0);
}
```

2. Για την ανάγνωση του αρχείου

```
if(arxeio==NULL)
{
      printf("\nCannot open file."); exit(-1);
}
```

3. Αν το πλήθος των σεμαφόρων είναι θετικός αριθμός

```
if(sem_count<=0)
{
      printf("\nWrong semaphorenumber.");
      exit(-1);
}
```

4. Αν η πιθανότητα των σημαφόρων είναι θετικός αριθμός

```
if(down_probability<=0)
{
      printf("\nWrong down probability"); exit(-
      1);
}
```

5. Αν ο μέσος χρόνος διατήρησης κατάστασης σεμαφόρου είναι θετικός

```
if(mean_semaphore_hold_time<=0)
{
      printf("\nWrong mean semaphore holdtime");
      exit(-1);
}
```

6. Αν η δέσμευση μνήμης για τις διεργασίες ολοκληρώθηκε σωστά προτού κλείσει το αρχείο

```
if(array==NULL)
{
      fclose(arxeio);
      printf("\nCannot malloc.");
      exit(-1);
}
```

Έπειτα από όλες τις απαραίτητες δεσμεύσεις μνήμης και αρχικοποίησης των απαραίτητων τιμών καλείται η συνάρτηση:

```
PRp(array,processes_count,sarray,sem_count,down_probability,
mean_semaphore_hold_time);
```

Πριν την ανάλυση της συνάρτησης, σκόπιμη είναι η ανάλυση του κώδικα που προσομοιάζεται η κίνηση του σεμαφόρου.

```
up_sem(array, runs_next, sarray, mean_semaphore_hold_time);
```

```
down_sem(array,runs_next,sarray,which_sem,mean_semaphore_hold_
time);
```

Κάθε από τις παραπάνω συναρτήσεις που καλούνται δέχονται ως είσοδο το δείκτη του πίνακα των διεργασιών, τη διεργασία που θα εκτελεστεί μετά, το δείκτη του πίνακα των σεμαφόρων και το μέσο χρόνο διατήρησης σεμαφόρου. Κάθε τιμή αρχικοποιείται με τις κατάλληλες τιμές που ορίζονται.

Η συνάρτηση που κλήθηκε από το κεντρικό πρόγραμμα υλοποιείται με την λογική της PriorityScheduling χωρίς κληρονόμηση προτεραιότητας. Η συνάρτηση αυτή δέχεται ως είσοδο το δείκτη στο πίνακα διεργασιών, το πλήθος των διεργασιών, δείκτη στο πίνακα σημαφόρων, το πλήθος σημαφορών, την πιθανότητα που οι σεμαφόροι βρίσκονται σε κατάσταση «Down» και το μέσο χρόνο διατήρησης της κατάστασης που βρίσκεται ο σημαφόρος. Κάθε διεργασία τρέχει όταν είναι ενεργή, δεν είναι μπλοκαρισμένη από κάποιο σημαφόρο και έχει τη μεγαλύτερη προτεραιότητα:

```
if(array[i].arrival <=t && array[i].service_left > 0 &&
array[i].blocked == 'n')
{
      if(array[i].priority > max_priority)
      {
            runs_next = i;
            max_priority = array[i].priority;
      }
}
```

Όταν μία διεργασία μπορεί να εκτελεστεί , επιλεγεί για την δεδομένη χρονική στιγμή t, να εκτελέσει για το σεμαφόρο «UP» ή «Down».

Για «UP» του σημαφόρου πρέπει να έχει μηδενιστεί ο χρόνος που μπορεί να είναι δεσμευμένη διεργασία από σημαφόρο:

```
if(array[runs_next].semaphore_hold>=0)
{
      array[runs_next].countdown--;
      if(array[runs_next].countdown == 0)
      {
            up_sem(array,runs_next,sarray,mean_semaphore_hold_t
      ime);
      }
}
```

Ή για την συνάρτηση «Down» του σημαφόρου πρέπει η πιθανότητα να είναι μικρότερη από την πιθανότητα που έχει δοθεί από το χρήστη και αν η συνθήκη είναι αληθής τότε:

```
possibility_down = rand()%101;
if(possibility_down<down_probability)
{
      which_sem = rand()%sem_count;
      down_sem(array,runs_next,sarray,which_sem,mean_semaphore_
hold_time);
}
```

Κάθε διεργασία που δεν εκτελείται ικανοποιείται η παρακάτω συνθήκη και ο μετρητής των διεργασιών που δεν εκτελέστηκαν τη δεδομένη χρονική στιγμή αυξάνεται κατά 1:

```
if(array[i].arrival <=t && array[i].service_left > 0 &&
i!=runs_next && array[i].priority > array[runs_next].priority)
{
      (array[i].waiting_from_lower)++;
}
```

Εν συνεχεία, το κεντρικό πρόγραμμα εμφανίζει στο χρήστη να πληκτρολογήσει οτιδήποτε για την εκτέλεση της συνάρτησης:

```
PRpINH(array,processes_count,sarray,sem_count,down_probabilit,
mean_semaphore_hold_time);
```

Υλοποιείται Priority Scheduling με κληρονόμηση προτεραιοτήτων, όπου δέχεται ως είσοδο:

1. Δείκτη στο πίνακα των διεργασιών
2. Το πλήθος των διεργασιών
3. Δείκτη στο πίνακα των σημαφόρων
4. Το πλήθος των σημαφόρων
5. Την πιθανότητα που ένας σημαφόρος μπορεί βρίσκεται σε κατάσταση «Down»
6. Και τον χρόνο που μπορεί να διατηρηθεί ένας σημαφόρος στην ίδια κατάσταση

Για κάθε χρονική στιγμή t γίνονται οι παρακάτω έλεγχοι:

Πρώτα γίνεται ο έλεγχος για την εύρεση της διεργασίας που είναι ενεργή, μη μπλοκαρισμένη, περιμένει να εκτελεστεί και η προτεραιότητής είναι μεγαλύτερη της μεταβλητής που αποθηκεύεται η μέγιστη προτεραιότητας.

```
if(array[i].arrival <=t && array[i].service_left > 0 &&
array[i].blocked == 'n')
{
      if(array[i].dynamic_priority > max_priority)
      {
      runs_next = i; max_priority = array[i].dynamic_priority;
      }
}
```

Αν η διεργασία που εκτελείται ολοκληρωθεί τότε ο σεμαφόρος που απασχολείτο καλεί τη συνάρτηση για να εκτελέσει την εντολή «UP»:

```
if(array[runs_next].semaphore_hold>=0)
{
      up_sem(array,runs_next,sarray,mean_semaphore_hold_time);
}
```

Αν η διεργασία δεν έχει ολοκληρωθεί τότε μειώνεται κατά 1 ο μετρητής των επιτρεπόμενων βημάτων και αν ο μετρητής μηδενιστεί τότε ο σεμαφόρος καλεί τη συνάρτηση για την εντολή «UP» αλλιώς χτυπάει πιθανότητα από 0-100 και αν είναι κάτω της πιθανότητος που δόθηκε από το χρήστη τότε ο σημαφόρος εκτελεί μέσω της συναρτήσεως την εντολή «Down», άρα γίνονται οι παρακάτω έλεγχοι:

```
if(array[runs_next].semaphore_hold>=0)
{
      array[runs_next].countdown--;
      if(array[runs_next].countdown == 0)
      {
            up_sem(array, runs_next, sarray,
      mean_semaphore_hold_time);
      }
}
else
{
      possibility_down = rand()%101
      if(possibility_down<down_probability)
      {
            which_sem = rand()%sem_count;
            down_sem(array, runs_next, sarray, which_sem,
      mean_semaphore_hold_time);
      }
}
```

Τελευταίος έλεγχος είναι βοηθητικός για την δημιουργία των στατιστικών, διαβάζει όλες τις διεργασίες και αυξάνει τη μεταβλητή κατά 1 για κάθε μία που δεν εκτελείται, είναι ενεργή και η προτεραιότητα είναι μεγαλύτερη από της τρέχουσας διεργασίας.

```
if(array[i].arrival <=t && array[i].service_left > 0 &&
i!=runs_next && array[i].priority > array[runs_next].priority)
{
      (array[i].waiting_from_lower)++;
}
```

Το κεντρικό πρόγραμμα μετά από τις δύο συναρτήσεις υπολογίζει το μέσο χρόνο αναμονής από διεργασία με μικρότερη προτεραιότητα, όπως φαίνεται και παρακάτω:

```
for(i=0; i<proccess_number; i++)
{
      mxa += (array[i].waiting_from_lower);
}
mxa = mxa/(float)proccess_number;
```

Αφού αναλύθηκε ο κώδικας οι διαφορές ανάμεσα στην απλή υλοποίηση της μεθόδου σε συνδυασμό με τους σημαφόρους και στην υλοποίηση της μεθόδου με σημαφόρους που η προτεραιότητα κληρονομείται.

Οι διαφορά ανάμεσα στις δύο συναρτήσεις είναι η κληρονόμηση της προτεραιότητας που φαίνεται στο κομμάτι του κώδικα:

```
if(array[i].dynamic_priority > max_priority)
{
      runs_next = i;
      max_priority = array[i].dynamic_priority;
}
```

Εν αντιθέσει, όταν η προτεραιότητα δεν κληρονομείται:

```
if(array[i].priority > max_priority)
{
      runs_next = i;
      max_priority = array[i].priority;
}
```

Κεφάλαιο 4 – Αποτελέσματα και Συμπεράσματα

Αποτελέσματα

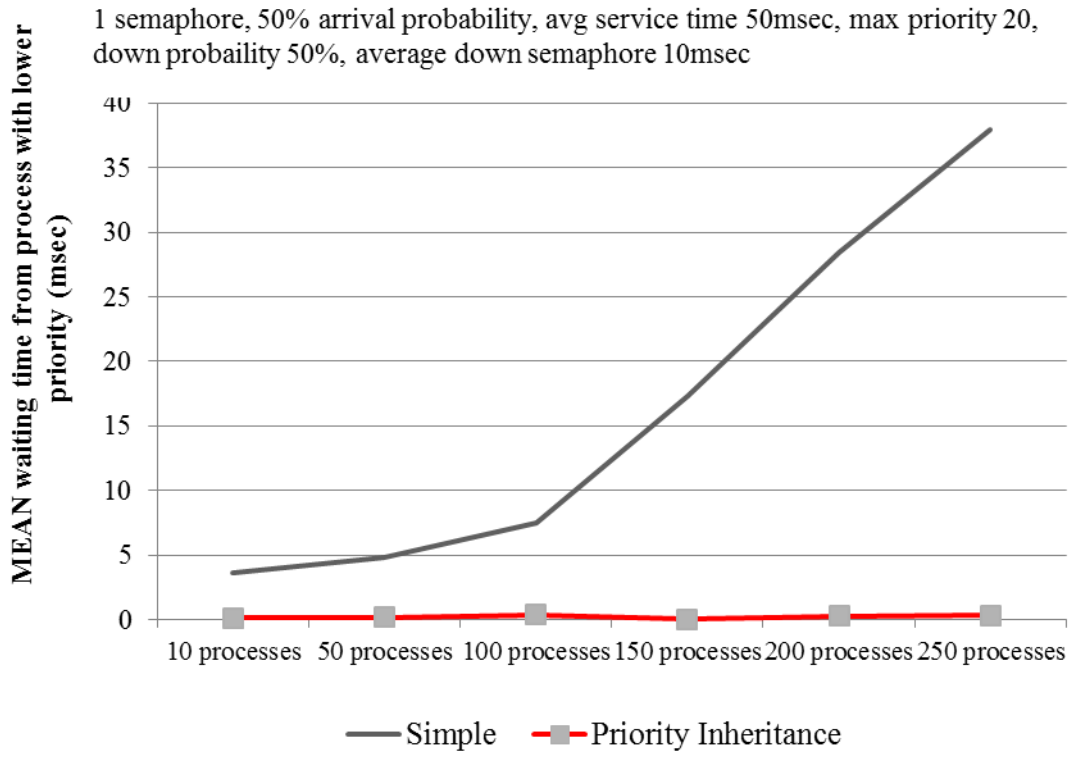
Αφού υλοποιήθηκε ο κώδικας και με τη βοήθεια του βοηθητικού προγράμματος create_process (κεφάλαιο 2) πάρθηκαν μετρήσεις του μέσου χρόνου αναμονής διεργασίας υψηλότερης προτεραιότητας, λόγω εκτέλεσης διεργασίας με χαμηλότερη προτεραιότητα, σε πολλαπλά αρχεία με τα εξής χαρακτηριστικά:

- Μεταβλητό πλήθος διεργασιών ένας σημαφόρος, 50% πιθανότητα εμφάνισης διεργασίας, ανά χρονικό κβάντο, μέγιστη προτεραιότητα διεργασίας 20 (δηλαδή, προτεραιότητες από 1 έως 20), πιθανότητα ενέργειας down σημαφόρου ίση με 50%, ανά χρονικό κβάντο , μέσος χρόνος εκτέλεσης διεργασιών 50 msec (ομοιόμορφη κατανομή με ελάχιστη τιμή 1 και μέγιστη τιμή 100) και μέσος χρόνος κατοχής σημαφόρου (σε κατάσταση Down) ίση με 10msec (ομοιόμορφη κατανομή με ελάχιστη τιμή 1 και μέγιστη τιμή 20):

	Simple	Priority Inheritance
10 processes	3,6	0,09
50 processes	4,76	0,16
100 processes	7,51	0,34
150 processes	17,29	0,20
200 processes	28,42	0,24
250 processes	37,93	0,32

Πίνακας 4.1 – Μέσοι χρόνοι αναμονής διεργασιών υψηλότερης προτεραιότητας, λόγω εκτέλεσης διεργασιών χαμηλότερης προτεραιότητας, για μεταβλητό πλήθος διεργασιών

Από τις τιμές μπορούμε να παρατηρήσουμε ότι ενώ στην απλή περίπτωση (χωρίς κληρονόμηση προτεραιοτήτων) ο χρόνος αναμονής κλιμακώνεται (αρνητικά) ανάλογα με το πλήθος των διεργασιών του συστήματος, στη δεύτερη περίπτωση (κληρονόμηση προτεραιοτήτων) ο χρόνος αυτός παραμένει (σχεδόν) σταθερός. Ο λογος που ο χρόνος αυτός είναι μη μηδενικός είναι ότι στην περίπτωση αυτή, ακόμα και με την κληρονόμηση προτεραιοτήτων, η διεργασία υψηλότερης προτεραιότητας, θα πρέπει να περιμένει τη διεργασία χαμηλότερης προτεραιότητας, που της πρόλαβε τον σημαφόρο (down), να τον ανεβάσει πάλι.

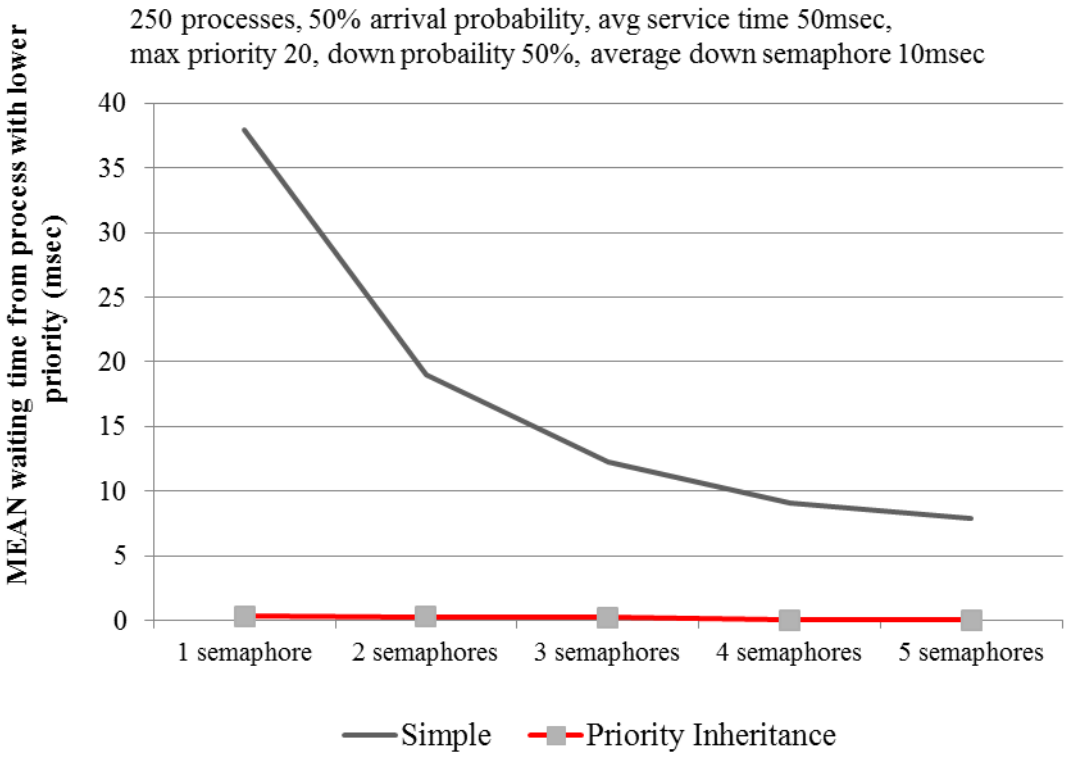


Εικόνα 4.1 Αποτέλεσμα εκτέλεσης με πολιτική Priority Scheduling Προεκτοπιστική και Κληρονόμηση Προτεραιοτήτων για μεταβλητό πλήθος διεργασιών

- Μεταβλητό πλήθος σημαφόρων 250 διεργασίες, 50% πιθανότητα εμφάνισης διεργασίας, ανά χρονικό κβάντο, μέγιστη προτεραιότητα διεργασίας 20 (δηλαδή, προτεραιότητες από 1 έως 20), πιθανότητα ενέργειας down σημαφόρου ίση με 50%, ανά χρονικό κβάντο , μέσος χρόνος εκτέλεσης διεργασιών 50 msec (ομοιόμορφη κατανομή με ελάχιστη τιμή 1 και μέγιστη τιμή 100) και μέσος χρόνος κατοχής σημαφόρου (σε κατάσταση Down) ίση με 10msec (ομοιόμορφη κατανομή με ελάχιστη τιμή 1 και μέγιστη τιμή 20):

	Simple	Priority Inheritance
1 semaphore	37,93	0,32
2 semaphores	18,95	0,25
3 semaphores	12,21	0,21
4 semaphores	9,1	0,022
5 semaphores	7,87	0,012
1 semaphore	37,93	0,32

Πίνακας 4.1 – Μέσοι χρόνοι αναμονής διεργασιών υψηλότερης προτεραιότητας, λόγω εκτέλεσης διεργασιών χαμηλότερης προτεραιότητας, για μεταβλητό πλήθος σημαφόρων



Εικόνα 4.2 Αποτέλεσμα εκτέλεσης με πολιτική Priority Scheduling Προεκτοπιστική και Κληρονόμηση Προτεραιοτήτων για μεταβλητό πλήθος σημαφόρων

Από τις τιμές μπορούμε να παρατηρήσουμε ότι ενώ στην απλή περίπτωση (χωρίς κληρονόμηση προτεραιοτήτων) ο χρόνος αναμονής κλιμακώνεται (θετικά) ανάλογα με το πλήθος των διεργασιών του συστήματος (όσο περισσότεροι σημαφόροι, τόσο μικρότερες τιμές), στη δεύτερη περίπτωση (κληρονόμηση προτεραιοτήτων) η κλιμάκωση αυτή είναι πολύ μικρότερη, αφού ξεκινάει από χαμηλά ήδη, και φυσικά πολύ μικρότερος από την πρώτη περίπτωση.

Συμπέρασμα

Σε κάθε περίπτωση, το σχήμα κληρονόμησης προτεραιοτήτων, μπορεί να εξασφαλίσει πολύ χαμηλούς χρόνους αναμονής διεργασιών υψηλότερης προτεραιότητας, λόγω εκτέλεσης διεργασιών χαμηλότερης προτεραιότητας, το οποίο αποδεικνύεται από τις εκτελέσεις των ανωτέρω προγραμμάτων.

Ακόμα και στην περίπτωση της κληρονόμησης, ο χρόνος αναμονής είναι μη μηδενικός, αλλά οφείλεται μόνο στο ότι μπορεί να τύχει διεργασία χαμηλότερης προτεραιότητα να κατεβάσει σημαφόρο (επιτυχής πράξη “up”) πριν τον κατεβάσει η διεργασία υψηλότερης προτεραιότητας και μόνο, άρα είναι πολύ χαμηλότερος (100 φορές χαμηλότερος, με βάση τις μετρήσεις μας, για υψηλές τιμές πλήθους διεργασιών και χαμηλές τιμές πλήθους σημαφόρων).

Παράρτημα

Στο παράρτημα περιέχεται ο κώδικας που υλοποιήθηκε για τη δημιουργία αρχείων εισόδου (Παράρτημα 1) , οι βασικές πολιτικές χρονοπρογραμματισμού (Παράρτημα 2), και στο Παραρτημα 3 η υλοποίηση προτεραιοτήτων μεταξύ των διεργασιών.

Παράρτημα 1

Στο παράρτημα αυτό δείχνει το κώδικα παραγωγής αρχείων εισόδου, ζητάει από το χρήστη το πλήθος των εγγραφών που θέλει να περιέχονται, την πιθανότητα εμφάνισης της κάθε διεργασίας, το μέγιστο χρόνο εκτέλεσης και τη μέγιστη προτεραιότητα.

Create_process.c

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

int main()
{
	int i, plithos, processes_count, arrival, service, priority;
	int probability, max_service_time, max_priority;
	char fileName[20], process_name[10];
	FILE* arxeio;

	printf("Onoma arxeiou: "); scanf("%s", fileName);

	printf("\nPlithos Processes: "); scanf("%d", &processes_count);
	if(processes_count<1) { printf("\nArnitiko plithos."); exit(-1);}

	printf("\nPithanotita Emfanisis diergasias: "); scanf("%d", &probability);
	if(probability<1 || probability>100) { printf("\nLathos Pithanotita."); exit(-1);}

	printf("\nMax Service Time: "); scanf("%d", &max_service_time);
	if(max_service_time<1) { printf("\nLathos max_service_time."); exit(-1);}

	printf("\nMax Priority: "); scanf("%d", &max_priority);
	if(max_priority<1) { printf("\nLathos max priority."); exit(-1);}

	arxeio=fopen(fileName,"w");
```

```
	if(arxeio==NULL) { printf("\nCannot create file."); exit(-1);}

	fprintf(arxeio,"%d\n",processes_count);
	for(i=0, plithos=0; plithos<processes_count; i++)
	{
		if(rand()%100 < probability)
		{
			sprintf(process_name, "pr%d\0", plithos);
			service = 1 + rand()%max_service_time;
			priority = 1 + rand()%max_priority;
		fprintf(arxeio,"%s %d %d %d\n",process_name, i, service, priority);
			plithos++;
		}
	}
	fclose(arxeio);

	return 1;
}
```

Παράρτημα 2

Οι βασικές μέθοδοι χρονοπρογραμματισμού που υλοποιούνται στο παράρτημα αυτό ζητούν ως είσοδο από το χρήστη το όνομα του αρχείου την πολιτική με τους συμβολισμούς:

1. FCFS για την πολιτική First Come First Served
2. SJFnp για την πολιτική Shortest Job First non-preemptive
3. SJFp για την πολιτική Shortest Job First preemptive
4. PRnp για την πολιτική Priority Scheduling non-preemptive
5. PRp για την πολιτική Priority Scheduling preemptive
6. RR (και το κβάντο) για την πολιτική Round Robin

Τέλος, υπολογίζει τους μέσους χρόνους αναμονής, εξυπηρέτσης, ολοκλήρωσης.

Main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "functions.h"


int main(int argc, char const *argv[])

/* arguments: FILENAME   POLICY {QUANTUM} (FOR R-R))*/

{

        int i, processes_count, q = 0, policy;
        FILE* arxeio;
        process* array;

        if(argc<3) { printf("\nYou gave too few arguments."); exit(0); }

        if(strcmp(argv[2], "FCFS") == 0) { policy = 1; }
        else if(strcmp(argv[2], "SJFnp") == 0) { policy = 2; }
        else if(strcmp(argv[2], "SJFp") == 0) { policy = 3; }
        else if(strcmp(argv[2], "PRnp") == 0) { policy = 4; }
        else if(strcmp(argv[2], "PRp") == 0) { policy = 5; }
        else if(strcmp(argv[2], "RR") == 0)
        {
                policy = 6;
                if(argc==3){printf("\nNo q given"); exit(0); }
                q = atoi(argv[3]); if(q<=0){ printf("\nq<=0"); exit(0);}
```

```
        }
        else { printf("\nYou gave wrong policy"); exit(0); }

        arxeio=fopen(argv[1],"r");
        if(arxeio==NULL) { printf("\nCannot open file."); exit(-1);}

        fscanf(arxeio,"%d",&processes_count);
        array=(process*)malloc(processes_count*sizeof(process));
        if(array==NULL) { fclose(arxeio); printf("\nCannot malloc."); exit(-1);}

        for(i=0; i<processes_count; i++)
        {
                if(feof(arxeio))
                {
                        printf("\nCorrupted File ");
                        fclose(arxeio);
                        break;
                }

fscanf(arxeio,"%s %d %d %d", array[i].name, &(array[i].arrival), &(array[i].service),
&(array[i].priority));
                array[i].service_left = array[i].service;
                array[i].waiting = 0;
                array[i].ends = -1;
        }
        fclose(arxeio);
        srand(time(NULL));
        printf("\n--- Reading ---");

        for(i=0; i<processes_count; i++)
        {
                printf("\n%s %d %d %d ", array[i].name, (array[i].arrival),
        (array[i].service), (array[i].priority)); }

        printf("\n\n--- Policy %s ---", argv[2]);
        if(policy == 1) FCFS(array, processes_count);
        if(policy == 2) SJFnp(array, processes_count);
        if(policy == 3) SJFp(array, processes_count);
        if(policy == 4) PRnp(array, processes_count);
        if(policy == 5) PRp(array, processes_count);
        if(policy == 6) RR(array, processes_count, q);

        print_stats(array, processes_count);
```

```
        free(array);
        return 1;
}
```

Function.h

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

typedef struct
{
	char name[8];
	int arrival, service, priority, service_left, waiting, ends;
	int state;  /* 0:hasn't come yet,  1:running,  2:ready,  3:blocked,  4:terminated */
}process;

void FCFS(process* array, int proccess_number); // FCFS
void SJFnp(process* array, int proccess_number); // SJF non preemptive
void SJFp(process* array, int proccess_number); // SJF preemptive
void PRnp(process* array, int process_number); // Priority non preemptive
void PRp(process* array, int process_number); // Priority preemptive
void RR(process* array, int process_number, int q); // PRq
void print_stats(process* array, int proccess_number);
```

Function.c

```
#include "functions.h"
#include <limits.h>

//*********************************************************************************
void FCFS(process* array, int process_number)//FCFS
{
	int i, t,runs_next, end, earlier_arrival;

	printf("\n\nGantt diagram: ");

	for(t=0; ; t++)
	{
		end = 1;
		for(i=0; i<process_number; i++)
		{
			//an estw kai mia exei service_time, den teleiwsame
			if(array[i].service_left > 0)
			{
				end = 0;
				break;
			}
		}
		if(end==1) return;

		earlier_arrival = INT_MAX;
		runs_next = -1;
		for(i=0; i<process_number; i++)
		{
			if(array[i].arrival <=t && array[i].service_left > 0)
			{
				if(array[i].arrival < earlier_arrival)
				{
					runs_next = i;
					earlier_arrival = array[i].arrival;
				}
			}
		}

		if(runs_next >= 0)
		{
			printf("%s ", array[runs_next].name);
```

```
                        (array[runs_next].service_left)--;
                        if(array[runs_next].service_left == 0)
                        {
                                array[runs_next].ends = t+1;
                        }

                        for(i=0; i<process_number; i++)
                        {
                        if(array[i].arrival<=t && array[i].service_left>0 && i!=runs_next)
                                (array[i].waiting)++;
                        }
                }
                else { printf("- "); }
        }
}

void SJFnp(process* array, int process_number)
{
        int i, t,runs_next, end, earlier_arrival, already_runs = -1, max_service;

        printf("\n\nGantt diagram: ");

        for(t=0; ; t++)
        {
                if(already_runs<0)
                {
                        end = 1;
                        for(i=0; i<process_number; i++)
                        {
                                // an estw kai mia exei service_time, den teleiwsame
                                if(array[i].service_left > 0)
                                {
                                        end = 0;
                                        break;
                                }
                        }
                        if(end==1) return;


                        max_service = INT_MAX;
                        runs_next = -1;
                        for(i=0; i<process_number; i++)
                        {
```

```
            if(array[i].arrival <=t && array[i].service_left > 0)
            {
                if(array[i].service_left < max_service)
                {
                    runs_next = i;
                    max_service = array[i].service_left;
                }
            }
        }
        else
        {
            runs_next = already_runs;
        }

        if(runs_next >= 0)
        {
            printf("%s ", array[runs_next].name);
            (array[runs_next].service_left)--;
            if(array[runs_next].service_left == 0)
            {
                array[runs_next].ends = t+1;
                already_runs = -1;
            }
            else already_runs = runs_next;

            for(i=0; i<process_number; i++)
            {
                if(array[i].arrival<=t && array[i].service_left>0 && i!=runs_next)
                    (array[i].waiting)++;
            }
        }
        else { printf("- "); already_runs = -1;}
}
}

void SJFp(process* array, int process_number) // SJF preemptive
{
        int i, t, runs_next, end, max_service;

        printf("\n\nGantt diagram: ");

        for(t=0; ; t++)
        {
```

```
end = 1;
for(i=0; i<process_number; i++)
{
	// an estw kai mia exei service_time, den teleiwsame
	if(array[i].service_left > 0)
	{
		end = 0;
		break;
	}
}
if(end==1) return;

max_service = INT_MAX;
runs_next = -1;
for(i=0; i<process_number; i++)
{
	if(array[i].arrival <=t && array[i].service_left > 0)
	{
		if(array[i].service_left < max_service)
		{
			runs_next = i;
			max_service = array[i].service_left;
		}
	}
}

if(runs_next >= 0)
{
	printf("%s ", array[runs_next].name);
	(array[runs_next].service_left)--;
	if(array[runs_next].service_left == 0)
	{
		array[runs_next].ends = t+1;
	}

	for(i=0; i<process_number; i++)
	{
	if(array[i].arrival<=t && array[i].service_left>0 && i!=runs_next)
		(array[i].waiting)++;
	}
}
else
{ printf("- "); }
```

```
	}
}

void PRnp(process* array, int process_number)
{
	int i, t,runs_next, end, earlier_arrival, already_runs = -1, max_priority;

	printf("\n\nGantt diagram: ");

	for(t=0; ; t++)
	{
		if(already_runs<0)
		{
			end = 1;
			for(i=0; i<process_number; i++)
			{
				if(array[i].service_left > 0)
				{
					end = 0;
					break;
				}
				// an estw kai mia exei service_time, den teleiwsame
			}
			if(end==1) return;


			max_priority = -1;
			runs_next = -1;
			for(i=0; i<process_number; i++)
			{
if(array[i].arrival<=t && array[i].service_left>0 && array[i].priority>max_priority)
				{
					runs_next = i;
					max_priority = array[i].priority;
				}
			}
		}
		else
		{
			runs_next = already_runs;
		}

		if(runs_next >= 0)
```

```
		{
			printf("%s ", array[runs_next].name);
			(array[runs_next].service_left)--;
			if(array[runs_next].service_left == 0)
			{
				array[runs_next].ends = t+1;
				already_runs = -1;
			}
			else already_runs = runs_next;

			for(i=0; i<process_number; i++)
			{
			if(array[i].arrival<=t && array[i].service_left>0 && i!=runs_next)
					(array[i].waiting)++;
			}
		}
		else
		{
			printf("- ");
			already_runs = -1;
		}
	}
}

void PRp(process* array, int process_number) // SJF preemptive
{
	int i, t, runs_next, end, max_priority;

	printf("\n\nGantt diagram: ");

	for(t=0; ; t++)
	{
		end = 1;
		for(i=0; i<process_number; i++)
		{
			// an estw kai mia exei service_time, den teleiwsame
			if(array[i].service_left > 0)
			{
				end = 0;
				break;
			}
		}
		if(end==1)
```

```
                return;


        max_priority = -1;
        runs_next = -1;
        for(i=0; i<process_number; i++)
        {
                if(array[i].arrival <=t && array[i].service_left > 0)
                {
                        if(array[i].priority > max_priority)
                        {
                                runs_next = i;
                                max_priority = array[i].priority;
                        }
                }
        }

        if(runs_next >= 0)
        {
                printf("%s ", array[runs_next].name);
                (array[runs_next].service_left)--;
                if(array[runs_next].service_left == 0)
                {
                        array[runs_next].ends = t+1;
                }

                for(i=0; i<process_number; i++)
                {
                if(array[i].arrival<=t && array[i].service_left>0 && i!=runs_next)
                                (array[i].waiting)++;
                }
        }
        else
        {
                printf("- ");
        }
    }
}

void RR(process* array, int process_number, int q)
{
    int i, t, *readyqueue, end, size = 0, runs_next, countdown = q, temp;
```

```
readyqueue = (int*)malloc(process_number*sizeof(int));
if(readyqueue==NULL)
{
	printf(“problem with memomry allocation”);
	return;
}

printf("\n\nGantt diagram: ");
for(t=0; ; t++)
{
	printf("\nt:%d, countdown: %d", t, countdown);
	end = 1;
	for(i=0; i<process_number; i++)
	{
		// an estw kai mia exei service_time, den teleiwsame
		if(array[i].service_left > 0)
		{
			end = 0;
			break;
		}
	}
	if(end==1)
		return;

	for(i=0; i<process_number; i++) // for new processes
	{
		if(array[i].arrival == t)
		{
			readyqueue[size] = i;
			size++;
		}
	}

	if(size==0)
	{
		printf("- ");
		continue;
	}

	runs_next = readyqueue[0];
	printf("%s ", array[runs_next].name);
	(array[runs_next].service_left)--;
	countdown--;
```

```
                if(array[runs_next].service_left == 0) // terminated
                {
                        array[runs_next].ends = t+1;
                        size--;
                        for(i=0; i<size; i++)
                                readyqueue[i] = readyqueue[i+1];
                        countdown = q; // for the next process
                }
                else if(countdown==0) // shift all processes to RR
                {
                        temp = readyqueue[0];
                        for(i=0; i<size-1; i++)
                                readyqueue[i] = readyqueue[i+1];
                        readyqueue[size-1] = temp;
                        countdown = q; // for the next process
                }

                for(i=0; i<process_number; i++)
                {
                        if(array[i].arrival<=t && array[i].service_left>0 && i!=runs_next)
                                (array[i].waiting)++;
                }
        }

        free(readyqueue);
}


void print_stats(process* array, int proccess_number)
{
        int i;
        float mxo = 0.0, mxa = 0.0, mxe = 0.0;

        for(i=0; i<proccess_number; i++)
        {
                printf("\n%s arrived at %d and finished at %d , waiting %d",
array[i].name, array[i].arrival, array[i].ends, array[i].waiting);
                mxo += (array[i].ends - array[i].arrival);
                mxa += (array[i].waiting);
                mxe += (array[i].service);
        }
```

```
        mxe = mxe/(float)proccess_number;
        mxa = mxa/(float)proccess_number;
        mxo = mxo/(float)proccess_number;
        printf("\nMxe: %.2f, Mxa: %.2f, Mxo: %.2f", mxe, mxa, mxo);
}
```

Παράρτημα 3

Στο παράρτημα αυτό πραγματοποιήθηκε η υλοποίηση προτεραιοτήτων μεταξύ των διεργασιών με προσομοίωση σημαφόρων, χωρίς κληρονόμηση προτεραιότητας και με κληρονόμηση. Ζητά από το χρήστη να δώσει:

1. Το όνομα του αρχείου
2. Το πλήθος των σημαφόρων
3. Την πιθανότητα που οι σημαφόροι θα μεταβαίνουν στην κατάσταση «Down»
4. Μέσος χρόνος διατήρησης κατάστασης σεμαφόρου
5. Κατά την ολοκλήρωση του πρώτου σκέλους (δηλαδή υλοποίηση προτεραιοτήτων χωρίς κληρονόμηση προτεραιότητας με προσομοίωση σημαφόρων) ζητά από το χρήστη να πληκτρολογήσει οποιοδήποτε πλήκτρο για να συνεχίσει την εκτέλεση του.

Τέλος, σε κάθε σκέλος του υπολογίζει το μέσο χρόνο αναμονής από διεργασία με χαμηλότερη προτεραιότητα.

Main_s.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "functions_s.h"

/* arguments: FILENAME, semaphore_number, down probability, mean semaphore hold time */
int main(int argc, char const *argv[])
{
	int i, processes_count, policy, sem_count, down_probability;
	int mean_semaphore_hold_time;
	char key[20];
	FILE* arxeio;
	process* array;
	semaphore* sarray;

	if(argc!=5)
	{
		printf("\nWrong argument number.");
		exit(0);
	}
```

```
arxeio=fopen(argv[1],"r");
if(arxeio==NULL)
{
	printf("\nCannot open file.");
	exit(-1);
}

sem_count = atoi(argv[2]);
if(sem_count<=0)
{
	printf("\nWrong semaphore number.");
	exit(-1);
}

down_probability = atoi(argv[3]);
if(down_probability<=0)
{
	printf("\nWrong down probability");
	exit(-1);
}

mean_semaphore_hold_time = atoi(argv[4]);
if(mean_semaphore_hold_time<=0)
{
	printf("\nWrong mean semaphore hold time");
	exit(-1);
}


fscanf(arxeio,"%d", &processes_count);
 array = (process*)malloc(processes_count*sizeof(process));
if(array==NULL)
{
	fclose(arxeio);
	printf("\nCannot malloc.");
	exit(-1);
}

for(i=0; i<processes_count; i++)
{
	if(feof(arxeio))
	{
```

```
                printf("\nCorrupted File ");
                fclose(arxeio);
                break;
        }
        fscanf(arxeio,"%s %d %d %d", array[i].name, &(array[i].arrival),
&(array[i].service), &(array[i].priority));
        array[i].service_left = array[i].service;
        array[i].waiting_from_lower = 0;
        array[i].ends = -1;
        array[i].semaphore_hold = -1;
        array[i].blocked = 'n';
         array[i].dynamic_priority = array[i].priority;
}
fclose(arxeio);
srand(time(NULL));
sarray = (semaphore*)malloc(sem_count*sizeof(semaphore));
if(sarray==NULL)
{
        fclose(arxeio);
        printf("\nCannot malloc semaphores.");
        exit(-1);
}
for(i=0; i<sem_count; i++)
{
        sarray[i].state = 1;
        sarray[i].owner = -1;
        sarray[i].plithos_block = 0;
        sarray[i].blocked = (int*)malloc(processes_count*sizeof(int));
        if(sarray[i].blocked ==NULL)
        {
                fclose(arxeio);
                printf("\nCannot malloc blocked semaphore lists.");
                exit(-1);
        }

printf("\n--- Reading ---");
for(i=0; i<processes_count; i++)
{
        printf("\n%s %d %d %d ", array[i].name, (array[i].arrival),
(array[i].service), (array[i].priority));
}
```

```
	/*****************************************************************/
	PRp(array, processes_count, sarray, sem_count, down_probability,
mean_semaphore_hold_time);
	print_stats(array, processes_count);

	printf("\nPress any key to run the inheritance code: "); scanf("%s", key);

	/* RE-initialize processes and semaphores */
	for(i=0; i<processes_count; i++)
	{
		array[i].service_left = array[i].service;
		array[i].waiting_from_lower = 0;
		array[i].ends = -1;
		array[i].semaphore_hold = -1;
		array[i].blocked = 'n';
	}

	for(i=0; i<sem_count; i++)
	{
		sarray[i].state = 1;
		sarray[i].owner = -1;
		sarray[i].plithos_block = 0;
	}

	PRpINH(array, processes_count, sarray, sem_count, down_probability,
mean_semaphore_hold_time);
	print_stats(array, processes_count);
	/*****************************************************************/

	for(i=0; i<sem_count; i++)
		free(sarray[i].blocked);

	free(array);
	free(sarray);
	return 1;
}
```

Functions_s.h

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

typedef struct blocked_process* deiktis;
typedef struct blocked_process
{
	int process;
	deiktis next;
}blocked_process;

typedef struct
{
	int state; // 0/1
	int owner; // which process owns it
	int* blocked; // list of blocked processes
	int plithos_block; // how many blocked
}semaphore;

typedef struct
{
	char name[8];
	int arrival, service, priority, service_left, waiting_from_lower, ends;
	int dynamic_priority;
	int semaphore_hold, countdown;
	char blocked; // 'y' or 'n'
}process;

// Priority preemptive
void PRp(process* array, int process_number, semaphore* sarray, int sem_count, int down_probability, int mean_semaphore_hold_time);

// Priority preemptive INHERITANCE
void PRpINH(process* array, int process_number, semaphore* sarray, int sem_count, int down_probability, int mean_semaphore_hold_time);

void up_sem(process* array, int runs_next, semaphore* sarray, int mean_semaphore_hold_time);

void down_sem(process* array, int runs_next, semaphore* sarray, int which_sem, int mean_semaphore_hold_time);
```

```
void print_stats(process* array, int proccess_number);
```

Functions_s.c

```
#include "functions_s.h"
#include <limits.h>

//*********************************************************************
void PRp(process* array, int process_number, semaphore* sarray, int sem_count, int down_probability, int mean_semaphore_hold_time) // SJF pre
{
	int i, t, runs_next, end, max_priority, possibility_down, which_sem;

	printf("\n\nGantt diagram: ");

	for(t=0; ; t++)
	{
		end = 1;
		for(i=0; i<process_number; i++)
		{
			// an estw kai mia exei service_time, den teleiwsame
			if(array[i].service_left > 0)
			{
				end = 0;
				break;
			}
		}
		if(end==1)
			return;

		printf("\nt=%d : ", t);

		max_priority = -1;
		runs_next = -1;
		for(i=0; i<process_number; i++)
		{
		if(array[i].arrival<=t && array[i].service_left>0 && array[i].blocked=='n')
			{
				if(array[i].priority > max_priority)
				{
					runs_next = i;
					max_priority = array[i].priority;
				}
			}
		}
```

```
if(runs_next >= 0)
{
	printf("%s runs ", array[runs_next].name);
	(array[runs_next].service_left)--;

	if(array[runs_next].service_left == 0) // terminates
	{
		array[runs_next].ends = t+1;
		printf(" and terminates ");
		// it must release the semaphore (if he has one)
		if(array[runs_next].semaphore_hold>=0)
		{
	printf(" ups semaphore %d ", array[runs_next].semaphore_hold);
	up_sem(array, runs_next, sarray, mean_semaphore_hold_time);
		}
	}
	else
	{
		if(array[runs_next].semaphore_hold>=0)
		{
			array[runs_next].countdown--;
			if(array[runs_next].countdown == 0)
			{
	up_sem(array, runs_next, sarray, mean_semaphore_hold_time);
			}
		}
		else
		{
			possibility_down = rand()%101;
			if(possibility_down<down_probability)
			// down semaphore
			{
				which_sem = rand()%sem_count;
down_sem(array, runs_next, sarray, which_sem, mean_semaphore_hold_time);
			}
		}
	}

	for(i=0; i<process_number; i++)
	{
		if(array[i].arrival <=t && array[i].service_left > 0 &&
i!=runs_next && array[i].priority > array[runs_next].priority)
```

```
                                        (array[i].waiting_from_lower)++;
                        }
                }
                else { printf("- "); }
        }
}

void PRpINH(process* array, int process_number, semaphore* sarray, int sem_count, int
down_probability, int mean_semaphore_hold_time)
{
        int i, t, runs_next, end, max_priority, possibility_down, which_sem;

        printf("\n\nGantt diagram: ");

        for(t=0; ; t++)
        {
                end = 1;
                for(i=0; i<process_number; i++)
                {
                        // an estw kai mia exei service_time, den teleiwsame
                        if(array[i].service_left > 0)
                        {
                                end = 0;
                                break;
                        }
                }
                if(end==1)
                        return;

                printf("\nt=%d : ", t);

                max_priority = -1;
                runs_next = -1;
                for(i=0; i<process_number; i++)
                {
                if(array[i].arrival<=t && array[i].service_left>0 & array[i].blocked == 'n')
                        {
                                if(array[i].dynamic_priority > max_priority)
                                {
                                        runs_next = i;
                                        max_priority = array[i].dynamic_priority;
                                }
                        }
```

```
	}

	if(runs_next >= 0)
	{
		printf("%s runs ", array[runs_next].name);
		(array[runs_next].service_left)--;

		if(array[runs_next].service_left == 0) // terminates
		{
			array[runs_next].ends = t+1;
			printf(" and terminates ");
			// it must release the semaphore (if he has one)
			if(array[runs_next].semaphore_hold>=0)
			{
		printf(" ups semaphore %d ", array[runs_next].semaphore_hold);
		up_sem(array, runs_next, sarray, mean_semaphore_hold_time);
			}
		}
		else
		{
			if(array[runs_next].semaphore_hold>=0)
			{
				array[runs_next].countdown--;
				if(array[runs_next].countdown == 0)
				{
		up_sem(array, runs_next, sarray, mean_semaphore_hold_time);
				}
			}
			else
			{
				possibility_down = rand()%101;

				if(possibility_down<down_probability)
				// down semaphore
				{
					which_sem = rand()%sem_count;
down_sem(array,runs_next,sarray,which_sem, mean_semaphore_hold_time);
				}
			}
		}

		for(i=0; i<process_number; i++)
		{
```

```
                if(array[i].arrival<=t && array[i].service_left>0 && i!=runs_next &&
        array[i].priority>array[runs_next].priority)
                                (array[i].waiting_from_lower)++;
                        }
                }
                else
                {
                        printf("- ");
                }
        }
}

void print_stats(process* array, int proccess_number)
{
        int i;
        float mxo = 0.0, mxa = 0.0, mxe = 0.0;

        for(i=0; i<proccess_number; i++)
        {
                printf("\n%s arrived at %d and finished at %d , waiting %d",
array[i].name, array[i].arrival, array[i].ends, array[i].waiting_from_lower);
                mxo += (array[i].ends - array[i].arrival);
                 mxa += (array[i].waiting_from_lower);
                 mxe += (array[i].service);
        }

        mxa = mxa/(float)proccess_number;
        printf("\nMxa_from_lower_priorities: %.2f", mxa);
}

void up_sem(process* array, int runs_next, semaphore* sarray, int
mean_semaphore_hold_time)
{
        int i, which_sem;

        which_sem = array[runs_next].semaphore_hold;
        array[runs_next].semaphore_hold = -1;
        array[runs_next].dynamic_priority = array[runs_next].priority;

        if(sarray[which_sem].plithos_block == 0)
        {
                sarray[which_sem].owner = -1;
                sarray[which_sem].state = 1;
```

```
	}
	else
	{
		sarray[which_sem].state = 0;
		sarray[which_sem].owner = sarray[which_sem].blocked[0];
printf(" , %d downs sem %d succesfully", sarray[which_sem].blocked[0], which_sem);
		(sarray[which_sem].plithos_block)--;

	array[sarray[which_sem].owner].dynamic_priority =
array[sarray[which_sem].owner].priority;

		for(i=0; i<sarray[which_sem].plithos_block; i++)
		{
			sarray[which_sem].blocked[i] = sarray[which_sem].blocked[i+1];
			if(array[sarray[which_sem].blocked[i]].priority >
array[sarray[which_sem].owner].dynamic_priority)
				array[sarray[which_sem].owner].dynamic_priority =
array[sarray[which_sem].blocked[i]].priority;
		}

		array[sarray[which_sem].owner].blocked = 'n';
		array[sarray[which_sem].owner].semaphore_hold = which_sem;
		array[sarray[which_sem].owner].countdown = 1+ rand()%
(2*mean_semaphore_hold_time);
	}
}

void down_sem(process* array, int runs_next, semaphore* sarray, int which_sem, int
mean_semaphore_hold_time)
{

	array[runs_next].semaphore_hold = which_sem;
	if(sarray[which_sem].state == 1)
	{
		printf(" , %d downs sem %d succesfully", runs_next, which_sem);
		sarray[which_sem].state = 0;
		sarray[which_sem].owner = runs_next;
		array[runs_next].blocked = 'n';
	array[runs_next].countdown = 1+ rand()%(2*mean_semaphore_hold_time);
	}
	else
	{
		array[runs_next].blocked = 'y';
```

```
            array[runs_next].semaphore_hold = which_sem;
        printf(" , %d downs sem %d and is being blocked", runs_next, which_sem);
        sarray[which_sem].blocked[sarray[which_sem].plithos_block] = runs_next;
            sarray[which_sem].plithos_block++;

        if(array[runs_next].priority>sarray[which_sem].owner)
        array[sarray[which_sem].owner].dynamic_priority = array[runs_next].priority;
        }
}
```

Βιβλιογραφία

1) ΣΥΓΧΡΟΝΑ ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ, 3η αμερικάνικη έκδοση, ANDREW S. TANENBAUM
2) Operating System Concepts, 6η έκδοση, John wiley and Sons
3) Τεχνολογία Υπολογιστικών Συστημάτων και Λειτουργικά Συστήματα, Παπακωνσταντίνου Γ., Τσανάκας Π., Κοζύρης Ν., Μανούσοπούλου Α., Ματζάκος Π.
4) ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ, ΙΩΑΝΝΗΣ Κ. ΚΑΒΟΥΡΑΣ, 2009.
5) Έννοια του Λειτουργικού συστήματος : https://el.wikipedia.org/wiki/%CE%9B%CE%B5%CE%B9%CF%84%CE%BF%CF%85%CF%81%CE%B3%CE%B9%CE%BA%CF%8C_%CF%83%CF%8D%CF%83%CF%84%CE%B7%CE%BC%CE%B1
6) Έννοια των Σημαφόρων: https://el.wikipedia.org/wiki/%CE%A3%CE%B7%CE%BC%CE%B1%CF%86%CF%8C%CF%81%CE%BF%CF%82_(%CF%85%CF%80%CE%BF%CE%BB%CE%BF%CE%B3%CE%B9%CF%83%CF%84%CE%AD%CF%82)