Τ.Ε.Ι ΠΕΛΟΠΟΝΝΗΣΟΥ

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ Τ.Ε.

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

«*Σχεδίαση και ανάπτυξη συστήματος προσομοίωσης συμπεριφοράς εφαρμογών, αναφορικά με τους αλγορίθμους αντικατάστασης σελίδας για σύγχρονα Λειτουργικά Συστήματα.*»

ARDELEAN ANDREEA LAURA

A.M. 2013044

Επιβλέπων καθηγητής

ΔΙΟΝΥΣΙΟΣ ΜΑΡΓΑΡΗΣ

Τ.Ε.Ι ΠΕΛΟΠΟΝΝΗΣΟΥ

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ Τ.Ε.

Ardelean Andreea Laura: A.M. 2013044

Επιβλέπων καθηγητής: Διονύσιος Μάργαρης

Εγκρίθηκε από την τριμελή επιτροπή την ….. Νοεμβρίου 2018

……	……	……
(Υπογραφή)	(Υπογραφή)	(Υπογραφή)

ΣΠΑΡΤΗ

Νοέμβριος 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):

……

Υπογραφή (Ολογράφως, χωρίς μονογραφή):

……

Ημερομηνία (Ημέρα – Μήνας – Έτος):

……

ΥΠΕΥΘΥΝΗ ΔΗΛΩΣΗ

Βεβαιώνω ότι είμαι συγγραφέας αυτής της πτυχιακής εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της, είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης έχω αναφέρει τις όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών ή λέξεων, είτε αυτές αναφέρονται ακριβώς, είτε παραφρασμένες.

Η συγκεκριμένη πτυχιακή εργασία προετοιμάστηκε από εμένα προσωπικά, ειδικά για τις απαιτήσεις του προγράμματος σπουδών του Τμήματος Μηχανικών Πληροφορικής Τ.Ε. του Τ.Ε.Ι. Πελοποννήσου.

Η συγγραφέας,

Ardelean Andreea Laura

ΕΥΧΑΡΙΣΤΙΕΣ

Για τη διεκπεραίωση της παρούσας πτυχιακής εργασίας, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή, Μάργαρη Διονύση, και ιδιαίτερα τους γονείς μου για την ψυχολογική και χρηματική υποστήριξη καθ' όλη την διάρκεια τον σπουδών μου.

ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ

ΠΕΡΙΛΗΨΗ

Στόχος της παρούσας πτυχιακής εργασίας είναι η σχεδίαση και ανάπτυξη συστήματος προσομοίωσης συμπεριφοράς εφαρμογών, αναφορικά με τους αλγορίθμους σελίδας, για σύγχρονα λειτουργικά συστήματα.

Η εφαρμογή που υλοποιήθηκε προσομοιώνει τη συμπεριφορά συστήματος διαχείρισης μνήμης βάσει πραγματικού ίχνους αναφορών.

Ο προσομοιωτής εκτελεί το μηχανισμό της εικονικής μνήμης βάσει ενός ανεστραμμένου πίνακα σελίδων, όπως στα πραγματικά Λειτουργικά Συστήματα.

Θα εξετάσουμε τους αλγορίθμους αντικατάστασης σελίδας:

- First In First Out (FIFO),
- Least Recently Used (LRU),
- Optimal (OPT) και
- Flush When Full (FWF).

1 ΘΕΩΡΗΤΙΚΟ ΥΠΟΒΑΘΡΟ

Στην ενότητα αυτή θα αναλύσουμε έννοιες, στο πλαίσιο των Λειτουργικών Συστημάτων, οι οποίες είναι χρήσιμες για την πλήρη κατανόηση της Πτυχιακής αυτή Εργασίας.

Πιο συγκεκριμένα, αναλύονται οι έννοιες:

- Εικονική Μνήμη,
- Σελιδοποίηση,
- Μονάδα Διαχείρισης Μνήμης,
- Πίνακας Σελίδων,
- Κρυφή Μνήμη,
- Κατακερματισμός,
- Ουρά

1.1 Εικονική μνήμη

Εικονική μνήμη (virtual memory) ονομάζεται η τεχνική διαχείρισης μνήμης που χρησιμοποιούν τα λειτουργικά συστήματα. Αυτή παρουσιάζει στις διεργασίες που εκτελούνται, παραπάνω μνήμη RAM από αυτή που διαθέτει το σύστημα, προκειμένου να διατηρείται η σταθερότητα τους.

Η εικονική μνήμη είναι αναγκαία διότι η εκτέλεση πολλών προγραμμάτων ταυτόχρονα έχει ως αποτέλεσμα να απαιτείται μνήμη που ξεπερνά το μέγεθος της φυσικής μνήμης.

Η βασική ιδέα είναι ότι το κάθε πρόγραμμα έχει το δικό του χώρο διευθύνσεων, ο οποίος διαιρείται σε μικρά κομμάτια που ονομάζονται σελίδες (pages). Κάθε σελίδα είναι ένα συνεχές εύρος διευθύνσεων.

Αυτές οι σελίδες χαρτογραφούνται στην μνήμη, αλλά δεν χρειάζονται να βρίσκονται όλες εκεί για να εκτελείται το πρόγραμμα.

Όταν ένα πρόγραμμα αναφέρεται σε ένα τμήμα του χώρου διευθύνσεων του στην φυσική μνήμη, το υλικό εκτελεί την απαραίτητη

χαρτογράφηση επιτόπου, και αν αναφέρεται σε τμήμα που δεν βρίσκεται στην φυσική μνήμη (RAM) τότε ειδοποιεί το λειτουργικό σύστημα να προσκομίσει το κομμάτι που λείπει .

Η εικονική μνήμη μπορεί να εφαρμοστεί θαυμάσια στα συστήματα πολυπρογραμματισμού, επιτρέποντας να βρίσκονται τμήματα από διάφορα προγράμματα στη μνήμη την ίδια χρονική στιγμή. Όσο ένα πρόγραμμα περιμένει να μεταφερθεί από το σκληρό δίσκο στη μνήμη κάποιο τμήμα του, η CPU μπορεί να εκχωρηθεί σε κάποια άλλη διεργασία.

1.2 Σελιδοποίηση

Τα περισσότερα συστήματα εικονικής μνήμης χρησιμοποιούν μια τεχνική η οποία ονομάζεται σελιδοποίηση (paging).

Σε κάθε υπολογιστή τα προγράμματα απευθύνονται σε ένα σύνολο διευθύνσεων μνήμης.

Οι διευθύνσεις δημιουργούνται με τη χρήση : αριθμοδεικτών, καταχτητών βάσης, καταχτητών τμημάτων, αλλά και με άλλους τρόπους.

Οι διευθύνσεις που δημιουργούνται από τα προγράμματά ονομάζονται εικονικές διευθύνσεις (virtual address) και συνθέτουν το χώρο εικονικών διευθύνσεων (virtual address space).

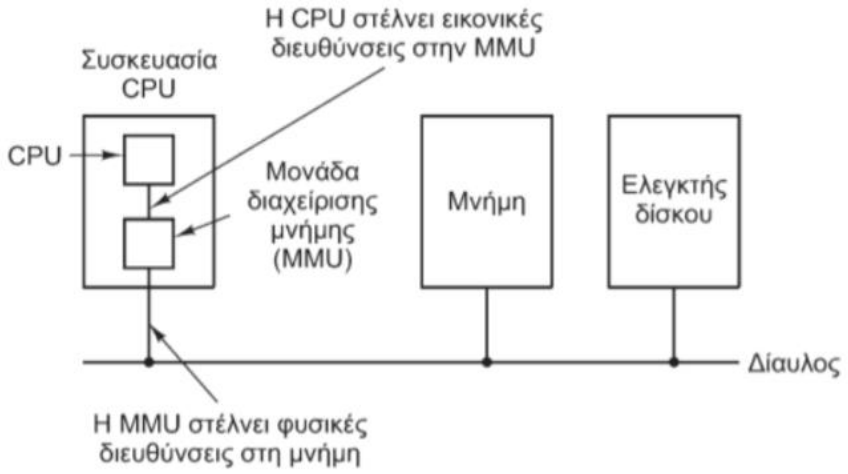
Σε υπολογιστές χωρίς εικονική μνήμη, οι εικονικές διευθύνσεις τοποθετούνται απευθείας στο δίαυλο της μνήμης (memory bus) που προκαλεί την ανάγνωση ή την εγγραφή της λέξης στην φυσική μνήμη που έχει την ιδιά διεύθυνση. Ενώ όταν χρησιμοποιείται εικονική μνήμη, οι εικονικές διευθύνσεις δεν τοποθετούνται απευθείας στο δίαυλο τις μνήμης , αλλά διοχετεύονται στην Μονάδα Διαχείρισης Μνήμης.

1.3 Μονάδα Διαχείρισης Μνήμης

Η Μονάδα Διαχείρισης Μνήμης (Memory Management Unit - MMU) χαρτογραφεί τις εικονικές διευθύνσεις σε διευθύνσεις της φυσικής μνήμης.

Όπως φαίνεται και στην Εικόνα 1. βρίσκεται στην κεντρική μονάδα επεξεργασίας (CPU), δηλαδή στο υλικό και όχι στο λογισμικό.

Άλλες δευτερεύουσες λειτουργίες είναι η προστασία μνήμης, ο έλεγχος της κρυφής μνήμης και η διαχείριση του δίαυλου (bus).



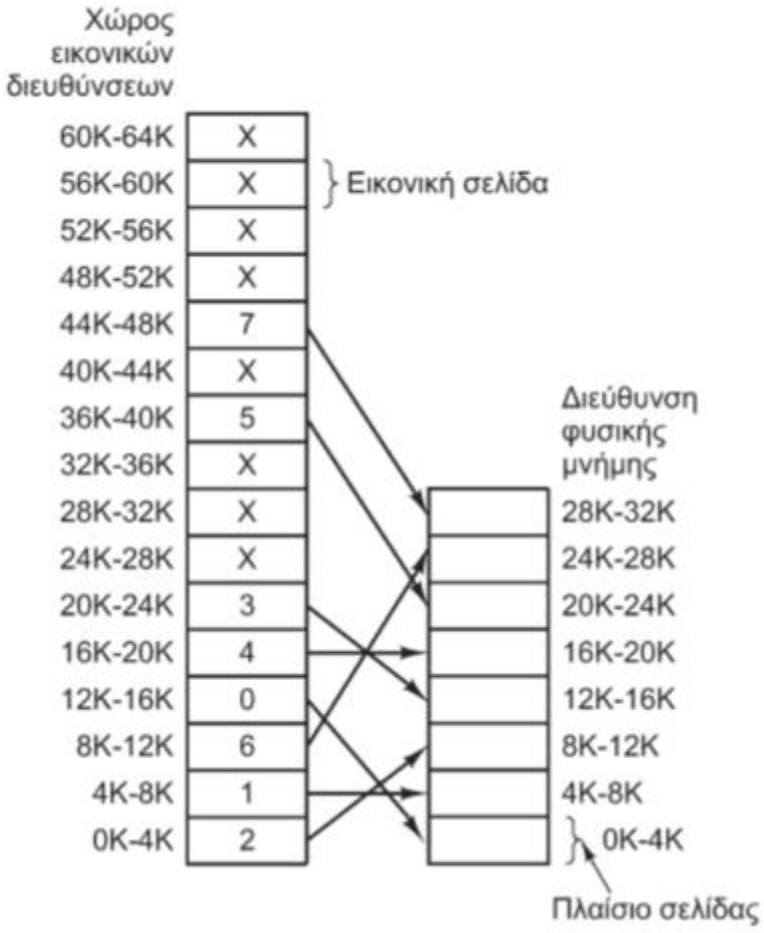
Εικόνα 1. Η εσωτερική λειτουργία της MMU

Ο χώρος των εικονικών διευθύνσεων διαιρείται σε μονάδες σταθερού μεγέθους οι οποίες ονομάζονται σελίδες (pages).

Οι αντίστοιχες μονάδες στην φυσική μνήμη ονομάζονται πλαίσια σελίδας (page frames).

Οι σελίδες και τα πλαίσια σελίδας έχουν, ως επί το πλείστον, το ίδιο μέγεθος. Όταν ένα πρόγραμμα καλεί μια εικονική διεύθυνση η μονάδα που ανέλαβε την χαρτογράφηση (MMU) ελέγχει αν η συγκεκριμένη διεύθυνση αντιστοιχίζεται στην φυσική μνήμη. Αν βρίσκεται τότε με βάση την αντιστοίχιση προσκομίζεται η ζητούμενη διεύθυνση. Αν όμως το πρόγραμμα προσπαθήσει να αναφερθεί σε μια σελίδα που δεν έχει

χαρτογραφηθεί στην φυσική μνήμη τότε η MMU διαπιστώνει ότι η ζητούμενη σελίδα δεν αντιστοιχίζεται στην φυσική μνήμη και αναγκάζει την CPU να καταφύγει σε παγίδευση (trap) του λειτουργικού συστήματος. Η παγίδευση αυτή ονομάζεται **σφάλμα σελίδας** (page fault). Το λειτουργικό σ' αυτή την περίπτωση επιλέγει ένα πλαίσιο σελίδας που έχει χρησιμοποιηθεί ελάχιστα και αποθηκεύει τα περιεχόμενά του στον δίσκο. Εν συνεχεία, προσκομίζει από τον δίσκο την ζητούμενη σελίδα και την τοποθετεί στο πλαίσιο σελίδας που μόλις ελευθερώθηκε. Τέλος επαναχαρτογραφείται η MMU και επανεκκινεί την εντολή που προκάλεσε την παγίδευση.



Εικόνα 2. Πίνακας σελίδων

1.3.1 Η εσωτερική λειτουργία της Μονάδας Διαχείρισης Μνήμης.

Για να καταλάβουμε καλυτέρα την λειτουργία στο εσωτερικό της MMU θα χρησιμοποιούμε το παρακάτω παράδειγμα :

Έστω ότι έχουμε μια εικονική διεύθυνση την 8196(0010000000000100 δυαδικό).

Η εισερχομένη εικονική διεύθυνση είναι 16bit, που διαιρείται σε:

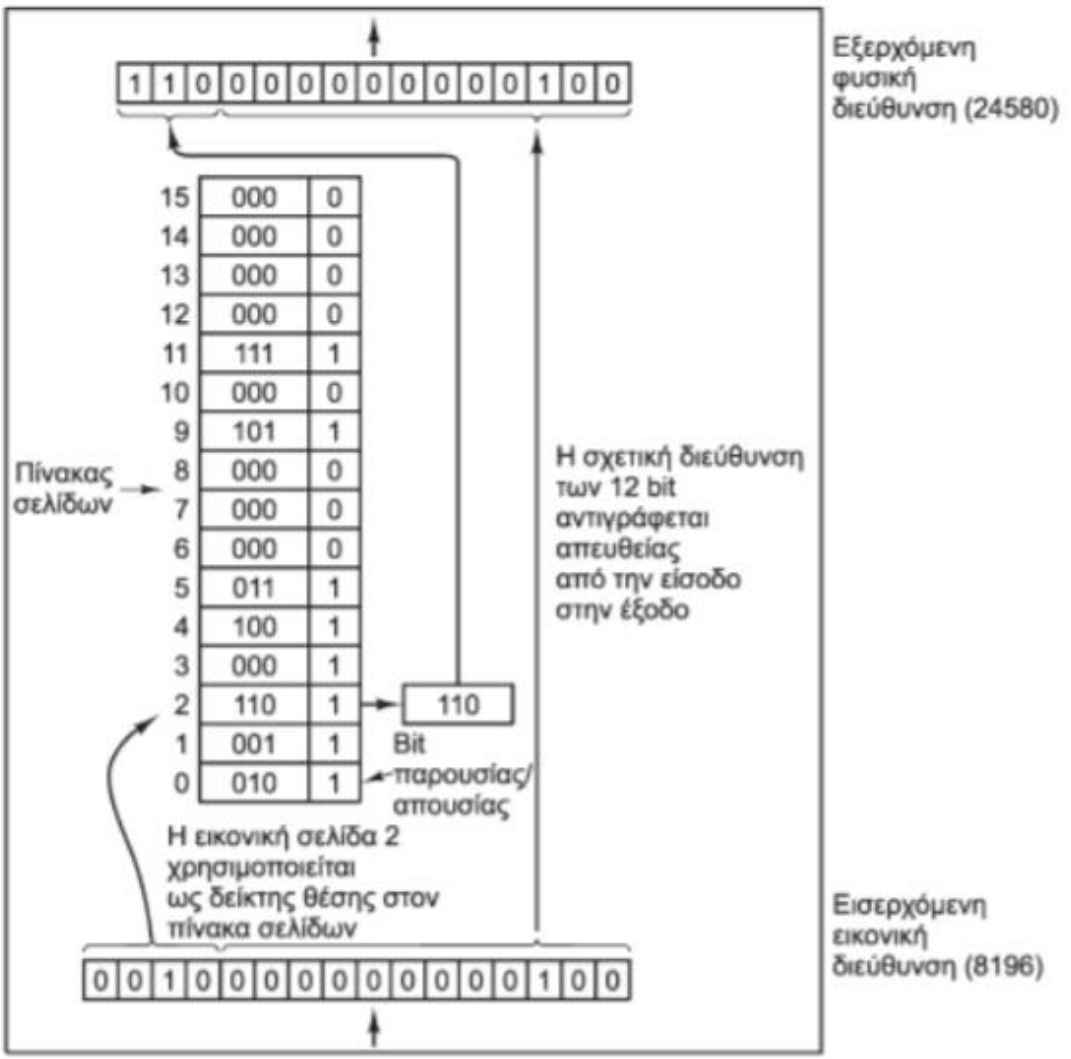
- έναν αριθμό σελίδας μήκους 4bit ,
- και μια σχετική διεύθυνση (offset) των 12bit .

Εφόσον έχουμε 4bit για τον αριθμό σελίδας αναφερόμαστε σε 16 σελίδες (2^4) , και μπορούμε να απευθυνθούμε με τα 12bit offset σε 4096 byte(2^12) μέσα σε μια σελίδα . Επόμενος ο πίνακας χαρτογράφησης έχει 16 θέσεις , περιέχει bit παρουσίας και δίνει 3 bit για το πλαίσιο .

Ο αριθμός σελίδας χρησιμοποιείται σαν αριθμοδείκτης θέσης στον πίνακα σελίδων (PageTable) και δείχνει τον αριθμό του πλαισίου σελίδας που αντιστοιχεί στην εικονική σελίδα .

Αν το bit παρουσίας/απουσίας είναι 0 τότε προκαλείτε σφάλμα , αν το bit είναι 1 τότε ο αριθμός πλαισίου αντιγράφει τα 3 bit του καταχωρητή εξόδου , μαζί με το offset και σχηματίζουν μια φυσική διεύθυνση των 15 bit.

Έτσι όπως φαίνεται και στην εικόνα 3.



Εικόνα 3. Η εσωτερική λειτουργία της MMU όταν υπάρχουν 16 σελίδες των 4 KB

1.4 Πίνακας σελίδων

Η αποστολή του πίνακα σελίδων είναι να χαρτογραφεί (να απεικονίζει ή να αντιστοιχίζει) εικονικές σελίδες σε πλαίσια σελίδων.

Η λειτουργία της MMU υλοποιείται με την χρήση των πινάκων σελίδων (page tables). Η χαρτογράφηση των εικονικών διευθύνσεων σε φυσικές διευθύνσεις μπορεί να συνοψιστεί ως εξής: η εισερχόμενη εικονική διεύθυνση διαιρείται σε:

- έναν αριθμό εικονικής σελίδας (bit υψηλής τάξης)
- και μια σχετική διεύθυνση (bit χαμηλής τάξης).

Ο αριθμός της εικονικής σελίδας χρησιμοποιείται ως αριθμοδείκτης θέσης στον πίνακα σελίδων προκειμένου να εντοπίζεται η καταχώριση της συγκεκριμένης εικονικής σελίδας. Από την καταχώρηση του πίνακα σελίδων υπολογίζεται ο αριθμός του πλαισίου σελίδας.

Ο πίνακας σελίδων είναι μια συνάρτηση, με όρισμα τον αριθμό εικονικής σελίδας και αποτέλεσμα τον αριθμό φυσικού πλαισίου.

Χρησιμοποιώντας το αποτέλεσμα αυτής της συνάρτησης, μπορούμε να αντικαταστήσουμε το πεδίο εικονικής σελίδας σε μια εικονική διεύθυνση με το πεδίο πλαισίου σελίδας, ώστε να σχηματίσουμε μια φυσική διεύθυνση μνήμης.

1.4.1 Δομή Καταχώρησης Πίνακα Σελίδων

Αριθμός πλαισίου - Δίνει τον αριθμό πλαισίου στον οποίο υπάρχει η τρέχουσα σελίδα που αναζητάμε. Ο αριθμός των απαιτούμενων bits εξαρτάται από τον αριθμό των πλαισίων .

Παρουσία/ Απουσία bit – Δηλώνει εάν μια συγκεκριμένη σελίδα που αναζητάμε είναι παρούσα ή απουσιάζει. Σε περίπτωση που δεν υπάρχει, αυτό ονομάζεται σφάλμα σελίδας. Ορίστηκε στο 0 αν η αντίστοιχη σελίδα δεν είναι στη μνήμη. Χρησιμοποιείται για τον έλεγχο σφάλματος σελίδας από το λειτουργικό σύστημα, για την υποστήριξη εικονικής

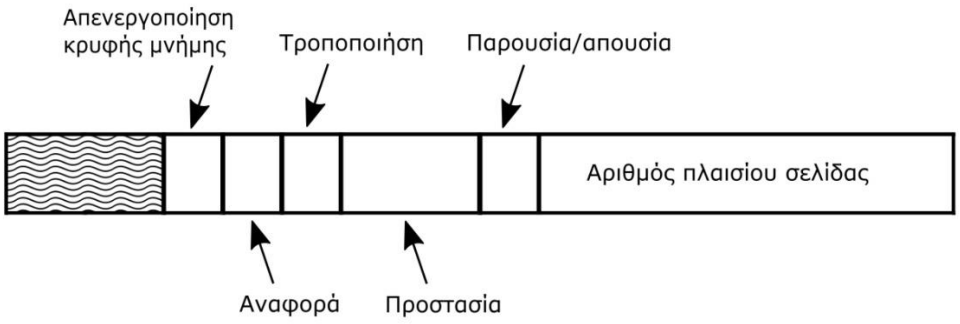
μνήμης. Μερικές φορές αυτό το bit είναι επίσης γνωστό ως έγκυρα / άκυρα κομμάτια.

Bit προστασίας - Το bit προστασίας λέει τι είδους προστασία θέλουμε σε αυτή τη σελίδα . Αυτό το πεδίο αποτελείται από 1 bit, οπού η τιμή 0 σημαίνει ότι επιτρέπεται ανάγνωση/εγγραφή, η τιμή 1 μονό για ανάγνωση. Μια εξεζητημένη έκδοση αποτελείται από 3 bit ,το πρώτο από τα οποία επιτρέπει ή όχι την ανάγνωση, το δεύτερο επιτρέπει ή όχι την εγγραφή και το τρίτο επιτρέπει ή όχι την εκτέλεση (executing)της σελίδας.

Τροποποίηση (Modified): Dirty bit δηλώνει ότι η σελίδα έχει τροποποιηθεί και σε περίπτωση εναλλαγής πρέπει να ξαναγραφεί στο δίσκο.

Αναφορά (Referenced): Δηλώνει ότι η σελίδα χρησιμοποιήθηκε είτε για ανάγνωση είτε για εγγραφή και παίρνει την τιμή 1 οποτεδήποτε γίνεται μια αναφορά στη συγκεκριμένη σελίδα. Χρησιμεύει στις αποφάσεις αντικατάστασης σελίδων.

(Απ)ενεργοποίηση κρυφής μνήμης (Caching): Κανονικά το Caching είναι on. Απενεργοποιείται μόνο για ειδικές σελίδες που αναφέρονται σε ενταμιευτές συσκευών Ε/Ε (I/O buffers) που απεικονίζονται στη μνήμη (memory mapped I/O).



Εικόνα 4. Μια τυπική καταχώριση του πίνακα σελίδων

1.4.2 Επιτάχυνση της σελιδοποίησης.

Σε οποιοδήποτε σύστημα σελιδοποίησης, υπάρχουν δύο σημαντικά ζητήματα που πρέπει να αντιμετωπιστούν:

1. Η χαρτογράφηση από την εικονική διεύθυνση στη φυσική πρέπει να είναι γρήγορη
2. Αν ο χώρος των εικονικών διευθύνσεων είναι μεγάλος, ο πίνακας σελίδων μπορεί να είναι και αυτός μεγάλος.

Οι τρόποι αντιμετώπισης αναλύονται στις επόμενες ενότητες.

1.5 Κρυφή μνήμη αναζήτησης μετάφρασης

Υπάρχουν πολλές μεθόδοι για την επιτάχυνση της σελιδοποίησης και το χειρισμό μεγάλων χωρών εικονικών διευθύνσεων. Ένας από αυτούς είναι η Κρυφή Μνήμη Αναζήτησης Μετάφρασης (Translation LookAside Buffer - TLB).

Στην ουσία, είναι μια μικρή συσκευή υλικού που χρησιμεύει στην χαρτογράφηση των εικονικών διευθύνσεων σε φυσικές χωρίς να χρησιμοποιεί τον πίνακα σελίδων .

Το TLB αναφέρεται συχνά και με τον όρο συνειρμική ή συσχετιστική μνήμη (Associative Registers).

Ας υποθέσουμε ότι μια εικονική διεύθυνση στέλνεται στην MMU, αν η καταχώρηση στον πίνακα σελίδων υπάρχει (ευστοχία – hit) γίνεται ανάκτηση του αριθμού πλαισίου και δημιουργείται η πραγματική διεύθυνση.

- Αν δε βρεθεί (αστοχία - miss), ο αριθμός σελίδας χρησιμοποιείται ως δείκτης στον πίνακα σελίδων της διεργασίας.
- Αν η ζητούμενη σελίδα υπάρχει εκεί, ανακτάται και το TLB ενημερώνεται ώστε να συμπεριλάβει τη νέα είσοδο σελίδας.
- Αν η σελίδα δεν υπάρχει στον πίνακα σελίδων (άρα και στην κύρια μνήμη) απορρέει ένα σφάλμα σελίδας (page fault).

1.5.1 Λειτουργία και διαχείριση της TLB με λογισμικό

Στις σύγχρονες μηχανές RISC όπως SPARC, MIRS και HPPA όλη η διαχείριση σελίδων υλοποιείται με λογισμικό. Η TLB φορτώνεται από το λειτουργικό σύστημα και όταν συμβαίνει κάποια αστοχία TLB προκύπτει απλώς σφάλμα TLB, και το πρόβλημα μεταβιβάζεται στο λειτουργικό σύστημα.

Υπάρχουν δύο ειδών αστοχίες, όταν χρησιμοποιείται διαχείριση TLB μέσω λογισμικού.

- Μια ήπια αστοχία(soft miss) που προκύπτει όταν η αναφερομένη σελίδα δεν βρίσκεται στην TLB αλλά στην μνήμη.
- Και η αυστηρή αστοχία(hard miss) που προκύπτει όταν η ίδια η σελίδα δεν υπάρχει στην μνήμη, αρά ούτε στην TLB.

1.6 Πίνακες σελίδων για μεγάλες μνήμες

Είδαμε μέχρι στιγμής ότι η TLB μπορεί να χρησιμοποιηθεί για την επιτάχυνση της μετάφρασης εικονικών διευθύνσεων σε φυσικές όταν χρησιμοποιείται η μέθοδος του πίνακα σελίδων στην μνήμη.

Ένα πρόβλημα που θα αντιμετωπίσουμε συχνά, είναι ότι θα χρειαστούμε πολύ μεγάλους χώρους εικονικών διευθύνσεων. Η λύση που έχει βρεθεί είναι είτε να χρησιμοποιούμε τους πολυεπίπεδους πίνακες σελίδων είτε τον ανεστραμμένο πίνακα.

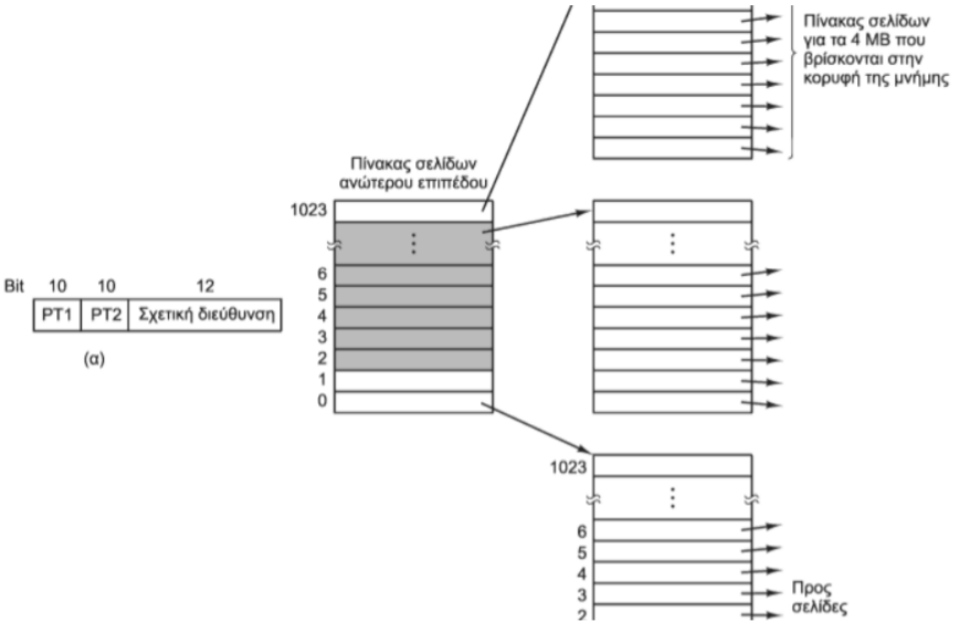
1.6.1 Πολυεπίπεδος πίνακας σελίδων

Σε αυτή την μέθοδο (MultiLevel PageTable) γίνεται τμηματοποίηση του πίνακα σελίδων έτσι ώστε να απαιτούνται λίγα διαμερίσματά του στην κεντρική μονάδα κάθε στιγμή.

Το μυστικό είναι ότι αποφεύγει τη συνεχή διατήρηση όλων των πινάκων σελίδων στη μνήμη, και η αποθήκευση μερικών από αυτούς γίνεται στο δίσκο.

Ένα μειονέκτημα είναι ότι το μέγεθος του αυξάνεται αναλογικά με το μέγεθος της ιδεατής μνήμης.

Μία εναλλακτική λύση είναι η χρήση ανεστραμμένων πινάκων σελίδων (inverted page tables).



Εικόνα 5. Πίνακες σελίδων δυο επίπεδων

1.6.2 Ανεστραμμένος πίνακα σελίδων

Πριν εξηγήσουμε πώς λειτουργεί ο ανεστραμμένος πίνακας σελίδων (Inverted PageTable), ας αναφέρουμε πρώτα γιατί ονομάζεται ανεστραμμένος.

Οι κανονικοί πίνακες σελίδων χαρτογραφούν εικονικές σελίδες σε πλαίσια φυσικών μνημών. Σε έναν ανεστραμμένο πίνακα σελίδας, για

κάθε κατειλημμένο πλαίσιο φυσικής μνήμης υπάρχει μία συνδεδεμένη εικονική σελίδα. Είναι ανεστραμμένη με την έννοια ότι εξετάζουμε τη χαρτογράφηση ξεκινώντας από ένα πλαίσιο φυσικής μνήμης πίσω σε μία εικονική σελίδα, αν και η πραγματική μετάφραση της διεύθυνσης αρχίζει με μια εικονική σελίδα μέχρι το φυσικό πλαίσιο μνήμης ακριβώς όπως ένα κανονικό table.

Η ιδέα πίσω από πίνακες ανεστραμμένων σελίδων είναι να υπάρχει ένας πίνακας μιας σελίδας στο επίπεδο του λειτουργικού συστήματος που δεν συνδέεται με κάποια συγκεκριμένη διαδικασία. Βασίζεται στην παρατήρηση ότι η CPU αναφέρει μόνο καταχωρήσεις σε εκείνες τις σελίδες που υπάρχουν ήδη στη μνήμη. Ο αριθμός των σελίδων που υπάρχουν στα πλαίσια φυσικής μνήμης είναι πολύ μικρότερος από τον συνολικό αριθμό των εικονικών σελίδων που βρίσκονται στο δίσκο. Έχοντας ένα μόνο πίνακα που χαρτογραφεί τα κατεχόμενα πλαίσια μνήμης σε εικονικές σελίδες, καταναλώνεται πολύ λιγότερος χώρος από το να έχεις έναν πίνακα σελίδων που είναι αρκετά μεγάλος για να ταιριάζει σε όλες τις εικονικές σελίδες για κάθε διαδικασία.

Δεδομένου ενός αναγνωριστικού διαδικασίας και ενός αριθμού εικονικής σελίδας, πρέπει να αναζητήσουμε τον ανεστραμμένο πίνακα σελίδας για μια αντιστοίχιση. Η γραμμική αναζήτηση απαιτεί χρόνο, οπότε η χρήση μιας αποδοτικής δομής δεδομένων, όπως ένας πίνακας κατακερματισμού, είναι ελκυστική.

1.6.2.1 Κατακερματισμένος Ανεστραμμένος Πίνακας Σελίδων (Hashed Inverted Page Table)

Κάνουμε Hash το αναγνωριστικό διαδικασίας pid(id) και τον αριθμό εικονικής σελίδας virtual page number(p).

Η συνάρτηση κατακερματισμού πίνακα ανεστραμμένων σελίδων μας δίνει ένα δείκτη στον πίνακα κατακερματισμού.

Χρησιμοποιούμε αυτό το δείκτη για να πάρουμε τον αριθμό φυσικού πλαισίου εάν η αποθηκευμένη διαδικασία D και ο αριθμός εικονικής

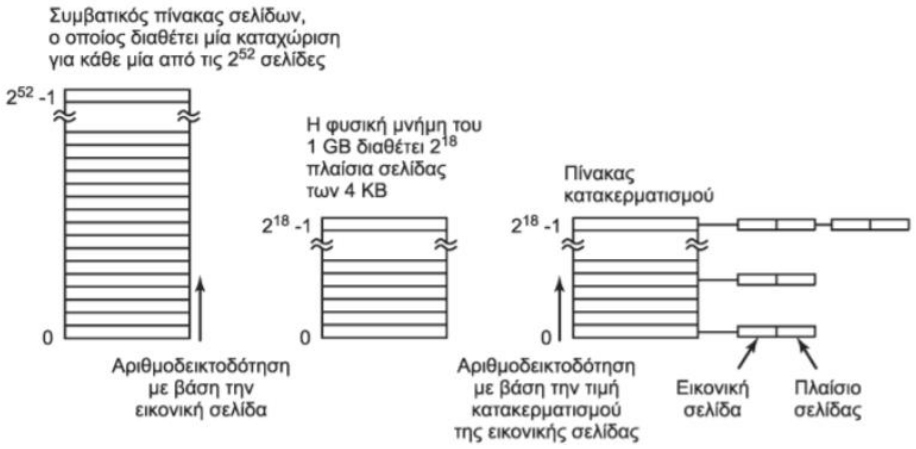
σελίδας σε εκείνη τη θέση δείκτη ταιριάζουν με το αναγνωριστικό διαδικασίας εισαγωγής και τον αριθμό εικονικής σελίδας.

Εάν δεν ταιριάζουν (διαφορετική διαδικασία D ή διαφορετικός αριθμός εικονικής σελίδας), τότε πρόκειται για σύγκρουση.

Σε αυτή την περίπτωση, ακολουθούμε το δείκτη στον επόμενο σύνδεσμο της αλυσίδας μέχρι να πάρουμε μια αντιστοίχιση.

Εάν όλες οι καταχωρίσεις στην αλυσίδα ελέγχονται χωρίς αντιστοιχία, αυτό σημαίνει λάθος ή σφάλμα σελίδας.

Στην πράξη, ο αριθμός των καταχωρήσεων στον πίνακα κατακερματισμού πρέπει να είναι μεγαλύτερος από τον αριθμό των φυσικών σελίδων, προκειμένου να μειωθούν οι συγκρούσεις του πίνακα κατακερματισμού.



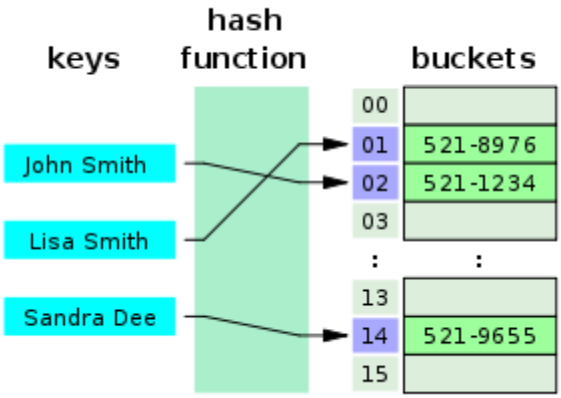
Εικόνα 6. Σύγκριση συμβατικού πίνακα σελίδων με ανεστραμμένο πίνακα σελίδων

1.7 Κατακερματισμός

Ο κατακερματισμός (hashing) είναι μια μέθοδος για τη φύλαξη στοιχείων με βάση ενός κλειδιού σε γραμμικές δομές δεδομένων (πίνακες, αρχεία) με στόχο τη γρήγορη ανεύρεσή τους. Βασίζεται στη χρήση μιας συνάρτησης απεικόνισης με πεδίο ορισμού το κλειδί των στοιχείων και πεδίο τιμών τους δείκτες της αντίστοιχης δομής δεδομένων.

1.7.1 Συνάρτηση Κατακερματισμού

Η Συνάρτηση Κατακερματισμού (Hash Function) είναι μια μαθηματική συνάρτηση που δέχεται ως είσοδο κάποιο δεδομένο τυχαίου μεγέθους και επιστρέφει ένα ακέραιο σταθερού μεγέθους αναπαράστασης. Το μέγεθος αυτό μπορεί να είναι από 32bit μέχρι 256bit ή και περισσότερα, ανάλογα με το λόγο χρήσης της συνάρτησης.



Εικόνα 7. Συνάρτηση κατακερματισμού

1.7.2 Πίνακας κατακερματισμού

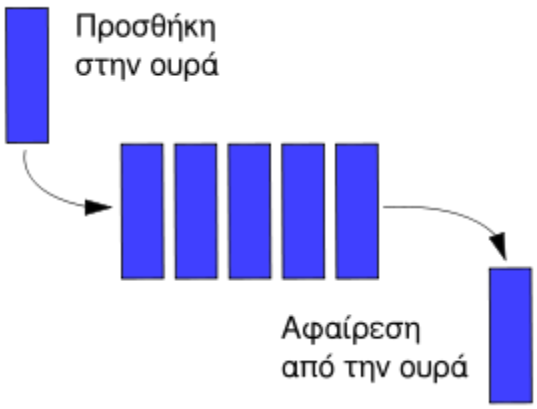
Οι συναρτήσεις κατακερματισμού χρησιμοποιούνται κυρίως σε πίνακες κατακερματισμού (Hash Tables), για γρήγορη εύρεση εγγραφών σε βάσεις δεδομένων.

Γενικότερα στην τεχνική του ευρετηρίου κατακερματισμού, η συνάρτηση κατακερματισμού μας δείχνει σε ποιο σημείο μέσα στην βάση βρίσκεται η εγγραφή. Στην πράξη μας δείχνει σε ποιο σημείο θα πρέπει να ξεκινήσουμε για την αναζήτηση.

Χαρακτηριστικό του πίνακα κατακερματισμού είναι ότι μπορεί να εκτελέσει σε σταθερό χρόνο, δηλαδή με πολυπλοκότητα O(1), τις λειτουργίες εισαγωγής, αναζήτησης και διαγραφής στοιχείων.

1.8 Ουρά

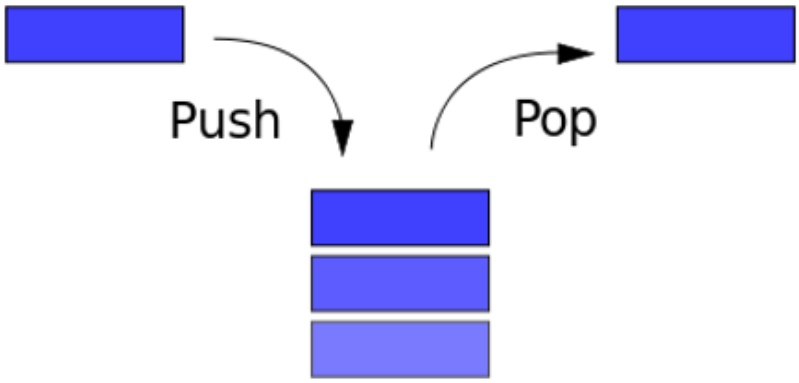
Η ουρά είναι ένας τύπος προσαρμογέα δοχείων που λειτουργούν με τη λογική πρώτος-μπαίνει πρώτος-βγαίνει (FIFO). Τα στοιχεία εισάγονται στο πίσω μέρος (τέλος) και διαγράφονται από μπροστά.



Εικόνα 8. Σχεδιασμός ουράς

Οι λειτουργίες που υποστηρίζονται από την ουρά είναι:

- empty () - Επιστρέφει αν η ουρά είναι κενή,
- size () - Επιστρέφει το μέγεθος της ουράς,
- front () - Επιστρέφει μια αναφορά στο πρώτο στοιχείο της ουράς,
- back () - Επιστρέφει αναφορά στο τελευταίο στοιχείο της ουράς,
- push (g) - Προσθέτει το στοιχείο 'g' στο τέλος της ουράς,
- pop () - Διαγράφει το πρώτο στοιχείο της ουράς.



Εικόνα 9. Σχεδιασμός στοίβας και ουράς

2 ΑΛΓΟΡΙΘΜΟΙ ΑΝΤΙΚΑΤΑΣΤΑΣΗΣ ΣΕΛΙΔΩΝ

Όταν συμβεί σφάλμα σελίδας, το λειτουργικό σύστημα πρέπει να διαλέξει μια σελίδα και να την αφαιρέσει από την μνήμη ώστε να εξοικονομήσει χώρο στη μνήμη για τη νέα σελίδα που θα φέρει. Αν η σελίδα που θα αφαιρεθεί έχει τροποποιηθεί, πρέπει να ξαναγραφτεί στο δίσκο. Στην συνέχεια φορτώνεται στη θέση της η νέα σελίδα.

Σκοπός των αλγόριθμων είναι να επιλέξουν την «πιο κατάλληλη» σελίδα που πρέπει να απομακρυνθεί από την κύρια μνήμη. Η σελίδα που θα αντικατασταθεί είναι αύτη που δεν θα χρειαστεί στο μέλλον για το μεγαλύτερο διάστημα, βέβαια οι αλγόριθμοι προσπαθούν να εκτιμήσουν κατά προσέγγιση ποια είναι η καλύτερη σελίδα, λαμβάνοντας υπ' όψη τους το ιστορικό της συμπεριφοράς των σελίδων μίας διεργασίας.

Αξίζει να σημειώσουμε ότι ανεξάρτητα από το ποιος αλγόριθμος αντικατάστασης σελίδων εφαρμόζεται, ορισμένα πλαίσια σελίδων στη μνήμη μπορεί να είναι κλειδωμένα. Τα κλειδωμένα πλαίσια έχουν πληροφορίες που χρειάζεται το λειτουργικό σύστημα για να λειτουργεί, όπως ο κώδικας του πυρήνα του Λ.Σ., σημαντικές δομές ελέγχου και η προσωρινή μνήμη.

Το κλείδωμα επιτυγχάνεται με την ύπαρξη ενός bit κλειδώματος στον πίνακα τμημάτων ή/και σελίδων το οποίο εξετάζει ο αλγόριθμος αντικατάστασης πριν επιλέξει μία σελίδα για να την απομακρύνει από την κύρια μνήμη.

Η απόδοση των αλγορίθμων αντικατάστασης κρίνεται από τα σφάλματα που παράγονται σε σχέση με το αλφαριθμητικό αναφοράς (reference string) που είναι μια ακολουθία αναφοράς σελίδων.

Στη συνέχεια του κεφαλαίου, παρουσιάζονται οι βασικοί αλγόριθμοι αντικατάστασης σελίδας, που μελετήθηκαν στην εργασία αυτή.

2.1 Ο αλγόριθμος αντικατάστασης σελίδας FIFO

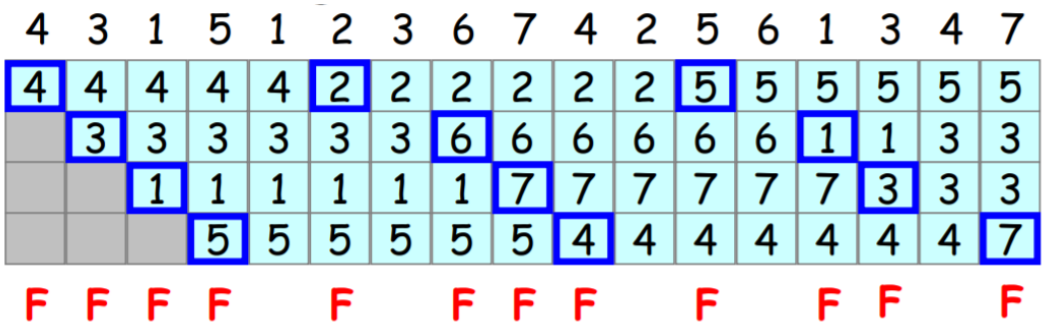
Στον αλγόριθμο Πρώτος μέσα πρώτος έξω (First-In, First-Out) το λειτουργικό σύστημα διατηρεί μια λίστα με τις σελίδες που βρίσκονται στη μνήμη.

Στην αρχή της λίστας βρίσκεται η παλαιότερη σελίδα και στην ουρά της λίστας η τελευταία σελίδα που εμφανίστηκε. Όταν μια σελίδα πρέπει να αντικατασταθεί δηλαδή γίνεται σφάλμα, η σελίδα που βρίσκεται στην αρχή της λίστας αφαιρείται.

Είναι ο απλούστερος αλγόριθμος αντικατάστασης σελίδας αλλά δεν εφαρμόζεται σήμερα στα λειτουργικά συστήματα.

Στο παρακάτω παράδειγμα έχουμε:

- 7 σελίδες
- 4 πλαίσια σελίδας



Εικόνα 10. Ο Αλγόριθμος FIFO

Αριθμός τον pagefaults : 12
Αριθμός hit : 5

2.2 Ο Αλγόριθμος αντικατάστασης σελίδας LRU

Ο Λιγότερος Πρόσφατα Χρησιμοποιούμενος (Least Recently Used - LRU) αλγόριθμος είναι ένας άπληστος αλγόριθμος όπου αντικαθιστάται η σελίδα που χρησιμοποιείται λιγότερο πρόσφατα. Η ιδέα βασίζεται στην τοποθεσία αναφοράς, η πιο πρόσφατα χρησιμοποιούμενη σελίδα δεν είναι πιθανή.

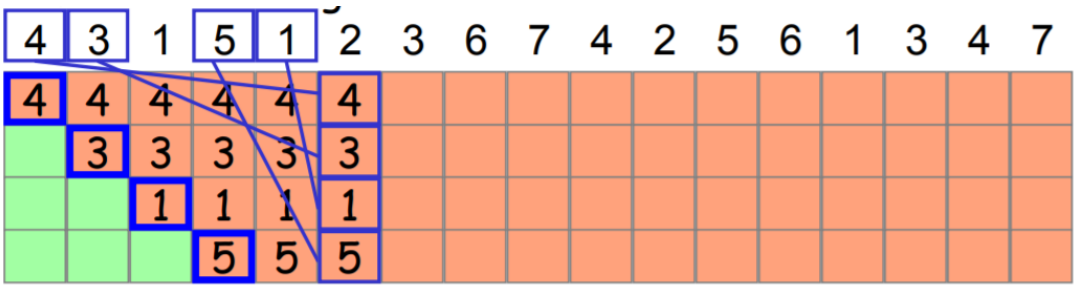
Ο αλγόριθμος υποθέτει ότι σελίδες που χρησιμοποιήθηκαν πρόσφατα θα ξαναχρησιμοποιηθούν σύντομα. Συνεπώς, όταν απαιτείται αντικατάσταση σελίδας, αντικαθίσταται η σελίδα που δεν έχει χρησιμοποιηθεί για το μεγαλύτερο χρονικό διάστημα.
Είναι εξαιρετικός αλγόριθμος αλλά παρουσιάζει δυσκολίες στην υλοποίηση σε σχέση με τον FIFO, καθώς απαιτείται καταγραφή της χρονικής στιγμής αναφοράς σε κάθε σελίδα και ενημέρωση σε κάθε αναφορά στη μνήμη.

I) Υλοποίηση λογισμικού: κρατείται λίστα των σελίδων στη μνήμη. Μετά από κάθε αναφορά στη μνήμη η αντίστοιχη σελίδα μετακινείται στη αρχή της λίστας. Αφαιρείται η τελευταία σελίδα. Η υλοποίηση αυτή είναι πολύ «ακριβή».

Στο παρακάτω παράδειγμα έχουμε :

- 7 σελίδες
- 4 πλαίσια σελίδας

Αρχική κατάσταση :



Εικόνα 11. Ο Αλγόριθμος LRU αρχική κατάσταση

Εικόνα 11. Ο Αλγόριθμος LRU αρχική κατάσταση

Τελική κατάσταση :

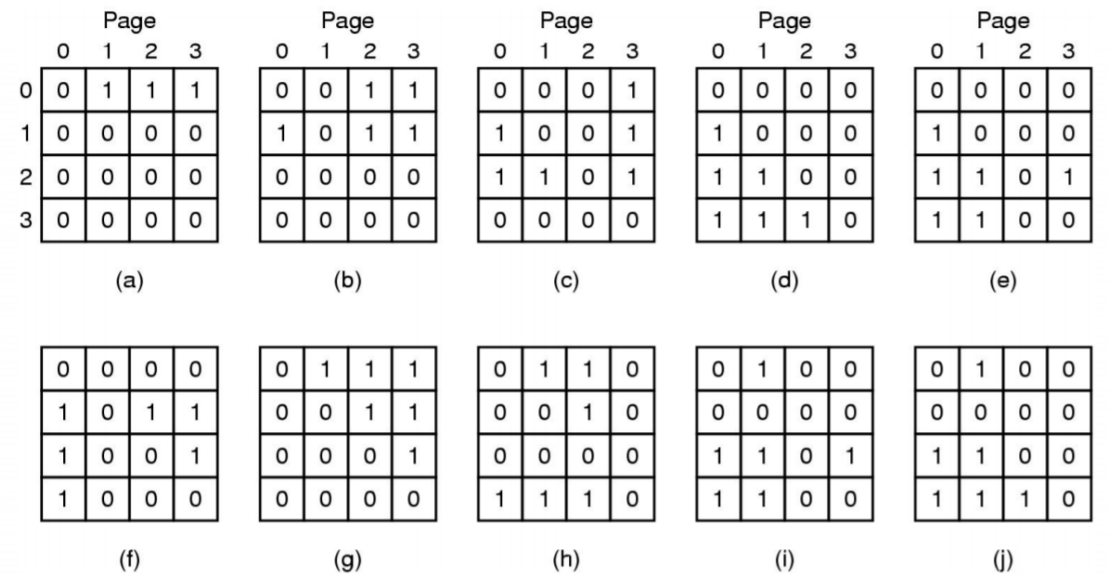
4	3	1	5	1	2	3	6	7	4	2	5	6	1	3	4	7
4	4	4	4	4	2	2	2	2	4	4	4	4	1	1	1	1
	3	3	3	3	3	3	3	3	3	2	2	2	2	3	3	3
		1	1	1	1	1	1	7	7	7	7	6	6	6	6	7
			5	5	5	5	6	6	6	6	5	5	5	5	4	4
F	F	F	F		F		F	F	F	F	F	F	F	F	F	F

Εικόνα 12. Ο Αλγόριθμος LRU τελική κατάσταση

Αριθμός pagefaults : 15
Αριθμός hit : 2

II) Υλοποίηση υλικού: Απαιτείται ένας καταχώρησης σε κάθε θέση του πίνακα σελίδων που ενημερώνεται από το ρολόι του συστήματος σε κάθε αναφορά στη μνήμη. Έστω μνήμη με n πλαίσια σελίδας , άμα έχουμε έναν πίνακά n x n οπού όλα τα bits παίρνουν αρχικά τιμή 0. Οπότε γίνεται αναφορά στο πλαίσιο σελίδας κ , το υλικό δίνει σε όλα τα bit της γραμμής κ την τιμή 1 ,και σε όλα τα bits της στήλης κ την τιμή 0. Όταν απαιτηθεί αντικατάσταση σελίδας, αφαιρείται η σελίδα με τη μικρότερη τιμή. Το μειονέκτημα είναι φυσικά ότι απαιτεί ειδικό υλικό.

Η λειτουργία του προσομοιωτή είναι χωρισμένη σε λογικά βήματα όπως φαίνεται παρακάτω:



Εικόνα 13. Ο Αλγόριθμος LRU

Ο αλγόριθμος με την χρήση πίνακά 4 πλαισίων σελίδας όταν οι αναφορές γίνονται με την σειράς 0, 1, 2 ,3 ,2 , 1, 0 ,3 , 2, 3.

2.3 Ο αλγόριθμος αντικατάστασης σελίδας OPT

Στον αλγόριθμο OPT (OPTIMAL – βέλτιστος), το λειτουργικό σύστημα αντικαθιστά τη σελίδα που δεν θα χρησιμοποιηθεί για το μεγαλύτερο χρονικό διάστημα στο μέλλον.

Η ιδέα είναι απλή, για κάθε αναφορά που κάνουμε παρακάτω:

- Εάν η αναφερόμενη σελίδα είναι ήδη παρούσα, μετρήστε το χτύπημα αύξησης (HIT).
- Αν δεν υπάρχει, βρείτε εάν μια σελίδα δεν αναφέρεται ποτέ στο μέλλον. Εάν υπάρχει μια τέτοια σελίδα, αντικαταστήστε αυτήν τη σελίδα με νέα σελίδα. Εάν δεν υπάρχει τέτοια σελίδα, βρείτε μια σελίδα που θα αναφέρεται πιο μακριά στο μέλλον. Αντικαταστήστε αυτή τη σελίδα με νέα σελίδα.

Η βέλτιστη αντικατάσταση σελίδας είναι τέλεια, αλλά δεν είναι δυνατή στην πράξη, καθώς το λειτουργικό σύστημα δεν μπορεί να γνωρίζει

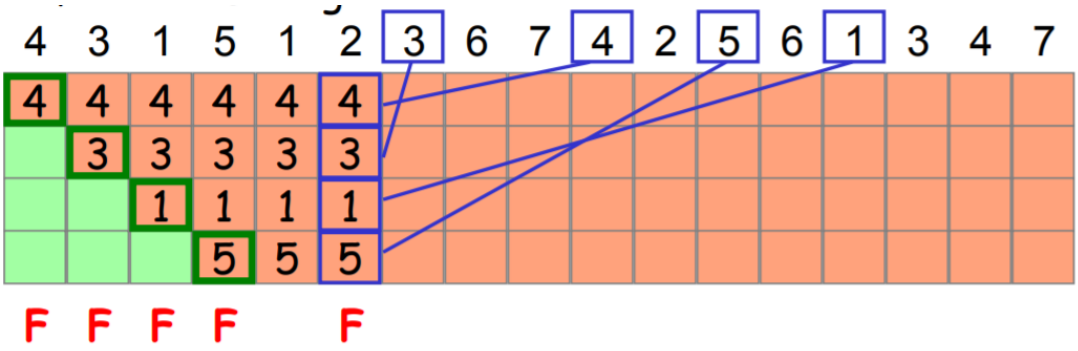
μελλοντικά αιτήματα. Η χρήση της αντικατάστασης της βέλτιστης σελίδας έχει σκοπό την δημιουργία ενός σημείου αναφοράς, ώστε να μπορούν να αναλυθούν άλλοι αλγόριθμοι αντικατάστασης.

Είναι γνωστός ως αλγόριθμος αντικατάστασης διόρθωσης για τη βέλτιστη πολιτική αντικατάστασης σελίδας του Bélády.

Στο παρακάτω παράδειγμα έχουμε :

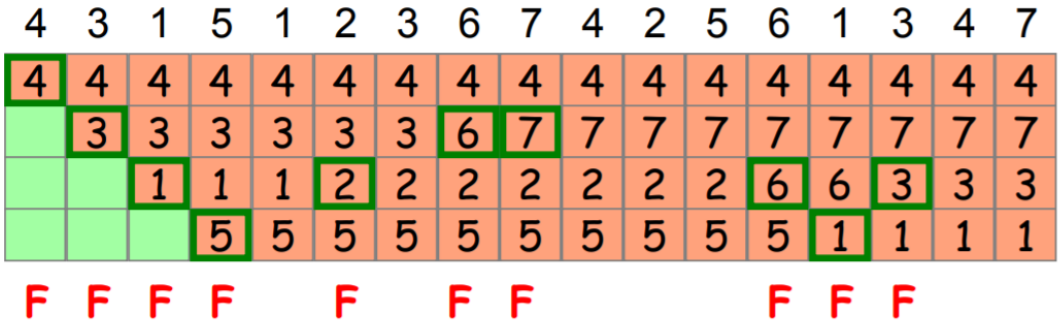
- 7 σελίδες
- 4 πλαίσια σελίδας

Αρχική κατάσταση :



Εικόνα 14. Ο Αλγόριθμος OPT αρχική κατάσταση

Τελική κατάσταση :



Εικόνα 15. Ο Αλγόριθμος OPT τελική κατάσταση

Αριθμός PageFaults : 15
Αριθμός hit : 2

2.4 Ο αλγόριθμος αντικατάστασης σελίδας FWF

Όταν η μνήμη cache είναι πλήρης και υπάρχει σφάλμα σελίδας, ο αλγόριθμος FWF (Flush When Full – Εκκαθάριση όταν γεμίσει η μνήμη) προκαλεί την εκκαθάριση της προσωρινής μνήμης.

Ας υποθέσουμε ότι αρχικά η προσωρινή μνήμη είναι κενή. Με ένα αίτημα για μια σελίδα μπορούν να παρουσιαστούν τρεις περιπτώσεις:

I. Η ζητούμενη σελίδα βρίσκεται σε κρυφή μνήμη: σε αυτήν την περίπτωση το αίτημα επιδίδεται χωρίς επιπλέον κόστος.
II. Παρουσιάζεται σφάλμα σελίδας και η μνήμη cache δεν είναι πλήρης: σε αυτή την περίπτωση η σελίδα εισάγεται στη μνήμη cache, δεν διώχνουμε καμία σελίδα.
III. Παρουσιάζεται σφάλμα σελίδας και η μνήμη cache είναι πλήρης: σε αυτή την περίπτωση, όλες οι σελίδες της προσωρινής μνήμης εξωθούνται, η μνήμη cache καθαρίζει.

Ο FWF δεν είναι αλγόριθμος σελιδοποίησης με βάση τη ζήτηση καθώς αδειάζει όλη τη γρήγορη μνήμη του κάθε φορά που αυτή είναι γεμάτη και συμβαίνει σφάλμα σελίδας. Πάντως, ο FWF μπορεί εύκολα να αναβαθμιστεί σε αλγόριθμο σελιδοποίησης με βάση τη ζήτηση. Αντί να αδειάζει τη γρήγορη μνήμη, μπορεί απλά να σημειώνει όλες τις σελίδες που υπάρχουν στη γρήγορη μνήμη και όποτε υπάρχει ένα νέο σφάλμα σελίδας, μπορεί να φέρει στη γρήγορη μνήμη τη νέα σελίδα αντικαθιστώντας μια σημειωμένη σελίδα.

Σε αυτό το κεφάλαιο παρουσιάζονται και αναλύονται τόσο τα δεδομένα του πειραματικού μέρους της εργασίας (σύνολα δεδομένων – datasets), καθώς και ο κώδικας που αναπτύχθηκε.

3.1 Τα αρχεία εισόδου .trace

Τα αρχεία μας bzip και gcc τα οποία έχουν τις καταχωρίσεις (διεργασίες) μας μέσα έχουν την κατάληξη .trace.

Το αρχείο εντοπισμού είναι αρχείο ιχνών (ή dump) που δημιουργεί η βάση δεδομένων της Oracle για να μας βοηθήσει να διαγνώσουμε και να επιλύσουμε προβλήματα λειτουργίας.

Κάθε διαδικασία διακομιστή και φόντου γράφει σε ένα αρχείο ιχνών. Όταν μια διαδικασία εντοπίσει ένα εσωτερικό σφάλμα, γράφει πληροφορίες σχετικά με το σφάλμα στο αρχείο ανίχνευσης.

3.2 Επεξήγηση του κώδικα

Ο κώδικας έχει υλοποιηθεί στη γλώσσα προγραμματισμού C++.

Ο κώδικας είναι χωρισμένος σε διαφορά αρχεία :

- trace.h
- main.cpp
- inverted_page_table.h
- status.h
- fifo.h
- fwf.h
- opt.h
- lru.h

Στη συνέχεια, θα αναλύσουμε το κάθε αρχείο ξεχωριστά.

3.2.1 Αρχείο trace.h

Θα αρχίσουμε από το Trace.h στο οποίο έχουμε ορίσει την δομή μιας διεργασίας.

Όλες τις διεργασίες τις έχουμε από τα αρχεία μας bzip.trace και gcc.trace.

Γνωρίζουμε ότι το κάθε αρχείο περιέχει ένα πλήθος γραμμών όπου παρουσιάζονται δεκαεξαδικές αναφορές μνήμης ακολουθούμενες από τους δείκτες R ή W, που προσδιορίζουν αν πρόκειται για εγγραφή ή ανάγνωση.

Επομένως με την δομή Trace περιγράφουμε πως είναι μια σελίδα.

Γνωρίζουμε επίσης ότι όλες οι σελίδες και τα πλαίσια έχουν μέγεθος 4 KB (4096 bytes).

Αρά με τα 4 πρώτα bit (virtual_page_number) μπορούμε να προσδιορίσουμε:

- 16 εικονικές σελίδες (2^4 = 16) και με τα
- 12 τελευταία bit (offset) τη σχετική διεύθυνση σε byte (2^12 = 4096).

Αναλυτική επεξήγηση στην σελ. (13)

Το pid (αναγνωριστικό διεργασίας) είναι ένας μοναδικός αριθμός αναγνώρισης της διεργασίας που σχετίζεται με τη λογική διεύθυνση.

Η καταχώριση στον ανεστραμμένο πίνακα σελίδων, περιέχει το αναγνωριστικό διεργασίας (pid) και τον αριθμό σελίδας (virtual_page_number).

3.2.1.1 Ο συνολικός κώδικας του αρχείου trace.h

```
#ifndef TRACE_H
#define TRACE_H
#pragma once
#define OFFSET_BITS12

Enum class Mode{ Read, Write };

Struct Trace {
    unsigned virtual_page_number;
    unsigned offset;
    unsigned pid;
    Mode mode;
};
#endif
```

3.2.2 Αρχείο main.cpp

Χρησιμοποιούμε ουρά για την υλοποίηση των αλγόριθμών.

Στην αρχή δημιουργούμε μια ουρά από Trace.

Αποτελείται από δυο παραμέτρους:

- Path: που θα μπει το κάθε trace
- Max_traces: ο μέγιστος αριθμός που μπορείς να βάλεις.

```
queue<Trace>load_traces (
    const string &path,
    unsigned max_traces = numeric_limits<unsignedint>::max()
)
```

Στην συνέχεια διαβάζουμε όσες διεργασίες μας έχει ορίσει ο χρήστης (max_traces) από τα αρχεία και τα βάζουμε στην ουρά.

```
   trace_file >> virtual_address >> mode && i < max_traces);
```

Μετατρέπουμε τις σελίδες από δεκαεξάδικο σε δυαδικό. Για την μετατροπή χρησιμοποιούμε το Stringstream, το οποίο είναι μια κλάση ροής που υπάρχει στην C++ και χρησιμοποιείται για την εκτέλεση εργασιών σε μια συμβολοσειρά.

```
stringstream ss;
      ss << hex << virtual_address;
      ss >> virtual_address_bin;
```

Συναρτήσεις :

```
     Bool still_have_traces(const vector<queue<Trace>>&programs)
```

Η συνάρτηση still_have_trace παίρνει έναν πίνακα από ουρές και ψάχνει άμα η ουρά έχει κάποια σελίδα μέσα και αντίστοιχα μας επιστέφει true ή false.

```
     queue<Trace>convert_to_single_queue(vector<queue<Trace>>programs,
unsigned q)
```

Μας επιστρέφει την τελική ουρά με τα Traces που υπάρχουν και στα 2 αρχεία, τα οποία και τα διαβάζουμε εναλλάξ.

Σε αυτή την συνάρτηση χρησιμοποιούμε τρεις έτοιμες συναρτήσεις που υπάρχουν στην ουρά ,την:

- front() που μας επιστρέφει ότι έχει μπροστά η ουρά
- push() που προσθέτει το trace στην ουρά
- pop() που το διαγράφει.

Όλο αυτό γίνεται με συνεχή επανάληψη (while) της συνάρτησης still_have_traces, μέχρι να διαβάζουμε όλες τις σελίδες που υπάρχουν στην ουρά.

Τέλος μας επιστρέφει την τελική ουρά.

```
    Return queue<Trace> result;
```

3.2.2.1 Ο συνολικός κώδικας του αρχείου main.cpp

```
#include<iostream>
#include<vector>
#include<queue>
#include<string>
#include<fstream>
#include<sstream>
#include<limits>
#include<cassert>

#include"./trace.h"
#include"./inverted_page_table.h"
#include"./algorithm/fifo.h"

Using namespace std;

queue<Trace>load_traces (
    const string &path,
    unsigned max_traces = numeric_limits<unsignedint>::max()
) {
    queue<Trace> result;

    ifstream trace_file(path);
    assert(trace_file.is_open());

    // Temp objects
    Trace current_trace;
    string virtual_address;
    unsigned virtual_address_bin;
    char mode;
```

```
        for (
        unsigned i = 0;
        (

            trace_file >> virtual_address >> mode

            && i < max_traces
        );
        ++i
    ) {
        stringstream ss;
        ss << hex << virtual_address;
        ss >> virtual_address_bin;

        current_trace.virtual_page_number = (virtual_address_bin >>
OFFSET_BITS);
        current_trace.offset = (virtual_address_bin & OFFSET_BITS);
        current_trace.mode = (mode == 'R') ? Mode::Read : Mode::Write;

        result.push(current_trace);
    }

    return result;
}


Bool still_have_traces(const vector<queue<Trace>>&programs) {

    for (constauto&program : programs) {
        if (program.size() >0) {
            returntrue;
        }
    }

    returnfalse;
}

queue<Trace>convert_to_single_queue(vector<queue<Trace>>
programs,unsigned q) {
    queue<Trace> result;

    while (still_have_traces(programs)) {

        for (unsigned prog = 0; prog <programs.size(); ++prog) {

            for (
                unsigned i = 0;
```

```
                (i < q) && (programs[prog].size() >0);
                ++i
            ) {
                Trace current_trace = programs[prog].front();
                current_trace.pid = prog;
                result.push(current_trace);
                programs[prog].pop();
            }
        }
    }

    return result;
}
```

```
Int main() {

    vector<queue<Trace>> programs {
        load_traces("../trace/bzip.trace", max_traces),
        load_traces("../trace/gcc.trace", max_traces)
    };

    auto all_traces = convert_to_single_queue(programs, q);

    fifo(all_traces, number_of_frames);
    lru(all_traces, number_of_frames);
    fwf(all_traces, number_of_frames);
    opt(all_traces, number_of_frames);

    return 0;
}
```

3.2.3 Αρχείο inverted_page_table.h

Κάθε εγγραφή στον ανεστραμμένο πίνακα σελίδων είναι ένα ζεύγος <αναγνωριστικό διεργασίας, αριθμός σελίδας>, όπου το αναγνωριστικό της διεργασίας (pid) αναλαμβάνει το ρόλο της αναγνώρισης στην θέσης της μνήμη. Όταν γίνεται αναφορά στη μνήμη, μέρος από την εικονική διεύθυνση, η οποία αποτελείται από το <αναγνωριστικό διεργασίας, αριθμός σελίδας>, δίνεται στο υποσύστημα της μνήμης.

- Αν βρεθεί μια εγγραφή, τότε παράγεται η φυσική διεύθυνση.

- Αν δε βρεθεί εγγραφή που να ταιριάζει στον πίνακα, τότε αυτό σημαίνει ότι δοκιμάστηκε πρόσβαση σε απαγορευμένη διεύθυνση.

Δομή

Χρησιμοποιούμε την δομή PageTableEntry για να εισάγουμε όλες τις σελίδες στον πίνακα μας.

Έχουμε :

- Το αναγνωριστικό διαδικασίας (pid)
- Το αριθμό σελίδας (virtual page number)
- Μια μεταβλητή τύπου boolean το οποίο ονομάσαμε dirty ,με την οποία δείχνουμε αν η σελίδα έχει τροποποιηθεί ή αναγνωσθεί (W or R).

```
Struct PageTableEntry {

     unsigned pid;
     unsigned virtual_page_number;
     bool dirty = false;
```

Το bool inline operator το χρησιμοποιούμε για την std::find.

```
Bool inline operator==(const PageTableEntry &rhs) {
  return    (pid    ==    rhs.pid)    &&    (virtual_page_number    ==
rhs.virtual_page_number);
       }
```

Αν το αναγνωριστικό διαδικασίας είναι ίσος με το pid που έχουμε στο πίνακα PageTableEntry και ο αριθμός σελίδας είναι ίσος με τον αριθμό

σελίδας στον πίνακα PageTableEntry τότε θα επιστέψουμε(return) την διεργασία αλλιώς έχουμε σφάλμα.

Συγκρίνουμε αν στο PageTableEntry οι δείκτες δείχνουν στην ιδιά θέση.

Στην συνέχεια δημιουργούμε ένα PageTableEntry από trace .

```
        // Create a PTE for trace
        PageTableEntry (const Trace &trace) {
            pid = trace.pid;
            virtual_page_number = trace.virtual_page_number;
            dirty = (trace.mode == Mode::Write);
        }

    }; //endifstructPageTableEntry
```

Ο inverted page table αποτελείται από ένα Vector που είναι ο πίνακας μας και ένα vector που είναι η λίστα η οποία κρατάει τα entries.

```
    Std::vector<std::vector<PageTableEntry>> inverted_page_table;
```

Η συνάρτηση κατακερματισμού (hash_function) μας δίνει έναν δείκτη στον πίνακα κατακερματισμού.

Αναλυτική περιγραφή σελίδες 19-20.

```
    Unsigned hash_function(Trace trace) {
      return (trace.pid + trace.virtual_page_number) % table_size;
  }
```

<u>Συναρτήσεις:</u>

```
    Bool search (Trace trace)
```

Με την συνάρτηση Search ψάχνουμε στο πίνακα κατακερματισμού μας αν έχουμε trace ή όχι.

Μας επιστέφει true or false εφόσον η συνάρτηση μας είναι τύπου boolean.

```
Void unload_from_ram (Trace trace)
```

Με την συνάρτηση unload from ram ξεφορτώνουμε από την μνήμη μια θέση.

Όταν πρέπει να «διώξουμε» ένα trace από την μνήμη το ψάχνουμε στον πίνακα κατακερματισμού και όταν το βρούμε ελέγχουμε αν το έχουμε τροποποιήσει, δηλαδή αν είναι dirty (Read or Write).

- Αν τον έχουμε τροποποιήσει (δηλαδή Write) τότε το γράφουμε στον δίσκο (CPU) και μετά το διαγράφουμε από την μνήμη μας.
- Αν δεν το έχουμε τροποποιήσει τότε απλώς ελευθερώνεται.

```
auto it = std::find(
         inverted_page_table[hash_value].begin(),
         inverted_page_table[hash_value].end(),
         trace
      );
```

Η std::find που χρησιμοποιεί ένα iterator ,που μας δίνει το trace που ψάχνουμε.

```
Void load_to_ram(Trace trace)
```

Με την συνάρτηση load_to_ram φέρνουμε μια trace στην μνήμη και με το push_back την σπρώχνουμε μέσα στο vector (PageTableEntry) .

3.2.3.1 Ο συνολικός κώδικας του αρχείου inverted_page_table.h

```
#ifndefSYSTEM_H
#defineSYSTEM_H
#pragmaonce

#include<vector>
#include<iostream>
#include<algorithm>
#include<experimental/optional>

#include"trace.h"
#include"stats.h"

Class InvertedPageTable {

    Struct PageTableEntry {

        unsigned pid;
        unsigned virtual_page_number;
        bool dirty = false;
        bool inline operator==(const PageTableEntry &rhs) {
            return   (pid   ==   rhs.pid)   &&   (virtual_page_number   ==
rhs.virtual_page_number);
        }

        PageTableEntry (const Trace &trace) {
            pid = trace.pid;
            virtual_page_number = trace.virtual_page_number;
            dirty = (trace.mode == Mode::Write);
        }
    };

    unsigned table_size;

    std::vector<std::vector<PageTableEntry>> inverted_page_table;

    unsigned hash_function(Trace trace) {
        return (trace.pid + trace.virtual_page_number) % table_size;
    }

public:

    InvertedPageTable (unsigned table_size) :
        table_size(table_size)  {// initializer list
        inverted_page_table.resize(table_size);
```

```
    }

    Bool search(Trace trace) {

        unsigned hash_value = hash_function(trace);

        return (
            std::find(  inverted_page_table[hash_value].begin(),
                inverted_page_table[hash_value].end(),
                PageTableEntry(trace)
            )
            != inverted_page_table[hash_value].end()
        );
    }

    Void unload_from_ram(Trace trace) {

        unsigned hash_value = hash_function(trace);

        auto it = std::find(
            inverted_page_table[hash_value].begin(),
            inverted_page_table[hash_value].end(),
            trace
        );

        if (it != inverted_page_table[hash_value].end()) {

            if (it->dirty) {
                write_on_disk();
            }

            inverted_page_table[hash_value].erase(it);
        }
    }

    Void load_to_ram(Trace trace) {

        inverted_page_table[hash_function(trace)].push_back(PageTableEn
try(trace));
    }
};

#endif// SYSTEM_H
```

3.2.4 Αρχείο status.h

Δημιουργούμε συναρτήσεις τύπου void, τις οποίες θα καλέσουμε στους αλγόριθμούς, και μας εκτυπώνουν το αντιστοιχώ μήνυμα.

3.2.4.1 Ο συνολικός κώδικας του αρχείου status.h

```
#ifndef STATS_H
#define  STATS_H
#pragma once

#include<iostream>

Void hit() { std::cout <<"Hit\n"; }
Void page_fault() { std::cout <<"Page Fault\n"; }
Void write_on_disk() { std::cout <<"Write on trace\n"; }

#endif// STATS_H
```

Σκοπός της Συνάρτησης:

Hit (): μας εκτυπώνει στην console τη λέξη 'Hit'

Page_Fault (): μας εκτυπώνει στην console την λέξη 'PageFault'

Write_on_disk (): μας εκτυπώνει στην console τη λέξη 'Write on trace'

3.2.5 Αρχείο fifo.h

Στον αλγόριθμο FIFO η παλαιότερη σελίδα βρίσκεται μπροστά στην ουρά. Όταν μια σελίδα πρέπει να αντικατασταθεί, τότε η σελίδα στο μπροστινό τμήμα της ουράς επιλέγεται για αφαίρεση.

Όταν έχουμε σφάλμα σελίδας τότε κάνουμε δυο ελέγχους:

- Αν το μέγεθος της ουράς μας είναι μικρότερο από τα number_of_frames που μας έδωσε ο χρήστης, τότε υπάρχει χώρος για να φορτώσουμε την σελίδα που ζητήσαμε.
- Αν δεν υπάρχει χώρος στην ουρά τότε αφαιρούμε την πρώτη σελίδα που βρίσκουμε, καλώντας την συνάρτηση unload_from_ram και την ξεφορτώνουμε από την μνήμη. Στη συνέχεια μέσω της χρήσης pop() την διαγράφουμε από την ουρά, και φορτώνουμε αυτή που έχουμε ζητήσει, με την συνάρτηση load_to_ram.

Τέλος προσθέτουμε στην ουρά μας το στοιχείο που έχουμε ζητήσει, με την συνάρτηση push.

Η συνάρτηση fifo παίρνει δυο παραμέτρους:

- queue<Trace>traces : μια ουρά από traces.
- number_of_frames: τον αριθμό των frames που θα το ορίσει ο χρήστης

```
void fifo(queue<Trace> traces, unsigned number_of_frames)
```

Σκοπός της Συνάρτησης:

Hit() : απλός διαβάσουμε την σελίδα που είδη υπάρχει μέσα στην μνήμη μας.

Page_Fault() : πρέπει να αφαιρούμε μια σελίδα, δηλαδή πρέπει να το ξεφορτώνουμε και από την μνήμη (IPT) και από την ουρά (fifo_queue).

3.2.5.1 Ο συνολικός κώδικας του αρχείου fifo.h

```
#ifndef FIFO_H
#define FIFO_H
#pragma once

#include <iostream>
#include <queue>
#include <string>
#include "../inverted_page_table.h"
#include "../trace.h"
#include "../stats.h"

using namespace std;

void fifo(queue<Trace> traces,unsigned number_of_frames) {

    InvertedPageTable IPT(number_of_frames);
    queue<Trace> fifo_queue;

    int hitcount = 0;
    int pagefaultcount = 0;

    while (!traces.empty()) {

        Trace trace = traces.front();
        traces.pop();
        cout << trace.a << ' ';
        if (IPT.search(trace)) {

            hit();
            hitcount++;

        } else {
            page_fault();

            pagefaultcount++;

            if (fifo_queue.size() < number_of_frames) {

                IPT.load_to_ram(trace);

            } else {
                IPT.unload_from_ram(fifo_queue.front());

                fifo_queue.pop();
```

```
                IPT.load_to_ram(trace);
            }
            fifo_queue.push(trace);

        }
    }
    cout << "Hits: " << hitcount << endl;
    cout << "PageFaults: " << pagefaultcount << endl;
}

#endif // FIFO_H
```

3.2.6 Το αρχείο fwf.h

Στον αλγόριθμο FlushWhenFull όταν έχουμε σφάλμα σελίδας τότε αδειάζουμε όλες τις διεργασίες από την μνήμη .

Κάνουμε τους έξης ελέγχους αν έχουμε σφάλμα σελίδας:

- Αν η μνήμη δεν είναι γεμάτη, τότε μας φέρνει την σελίδα.
- Αν η μνήμη είναι γεμάτη τότε διαγράφουμε όλη την ουρά και την μνήμη, και φέρνουμε την σελίδα που ζητήσαμε.

Η συνάρτηση FWF παίρνει δυο παραμέτρους:

- queue<Trace>traces: μια ουρά από traces.
- number_of_frames: τον αριθμό των frames που θα το ορίσει ο χρήστης.

```
Void fwf(queue<Trace>traces, unsigned number_of_frames)
```

Σκοπός της Συνάρτησης:

Όταν έχουμε Page_Fault() :

- Αν έχουμε χώρο στην ουρά, τότε βάζουμε τη σελίδα μέσα .

➢ Αν η ουρά είναι γεμάτη, τότε τη διαγράφουμε και βάζουμε το καινούργιο στοιχείο που ζητάμε μέσα.

Όταν έχουμε Hit() : απλώς διαβάζουμε τη σελίδα που ήδη υπάρχει μέσα στην μνήμη μας.

3.2.6.1 Ο συνολικός κώδικας της fwf.h

```
#ifndef FWF_H
#define FWF_H
#pragma once

#include <queue>
#include "../inverted_page_table.h"
#include "../trace.h"
#include "../stats.h"


using namespace std;

void fwf(queue<Trace> traces,unsigned number_of_frames){

    InvertedPageTable IPT(number_of_frames);
    queue<Trace> ff_queue;

    int pagefaultcount = 0;
    int hitcount = 0;

    while (!traces.empty()) {

        Trace trace = traces.front();
        traces.pop();
        cout << trace.a << ' ';
        if (IPT.search(trace)) {

            hit();
            hitcount++;

        }  else {
            page_fault();
```

```
            pagefaultcount++;

            if(ff_queue.size() == number_of_frames){

                while (!ff_queue.empty())
              {
                IPT.unload_from_ram(ff_queue.front());
                ff_queue.pop();
              }
            }
             IPT.load_to_ram(trace);
             ff_queue.push(trace);
        }



    }
     cout << "Hits: " << hitcount << endl;
     cout << "PageFaults: " << pagefaultcount << endl;

}

#endif // FWF_H
```

3.2.7 Το αρχείο lru.h

Με τον αλγόριθμο Least Recently Used (LRU) αντικαθιστούμε τη σελίδα που δεν έχει χρησιμοποιηθεί για το μεγαλύτερο χρονικό διάστημα. Μετά από κάθε αναφορά στη μνήμη, η αντίστοιχη σελίδα μετακινείται στην αρχή της ουράς και αφαιρείται η τελευταία. Με αυτό το τρόπο γνωρίζουμε ποια σελίδα έχει να χρησιμοποιηθεί για περισσότερο καιρό.

Σκοπός της Συνάρτησης:

Η συνάρτηση lru παίρνει δυο παραμέτρους:

- queue<Trace>traces: μια ουρά από traces.
- number_of_frames: τον αριθμό των frames που θα το ορίσει ο χρήστης.

```
Void lru(queue<Trace> traces, unsigned number_of_frames)
```

Όταν έχουμε Hit () :

- Μετακινούμε την σελίδα που ήδη υπάρχει μέσα στην ουρά μας, στην αρχή της ουράς.

Όταν έχουμε PageFault() κάνουμε δυο ελέγχους:

- Αν έχουμε χώρο στην ουρά, τότε βάζουμε την σελίδα μέσα

- Αν η ουρά είναι γεμάτη, τότε διαγράφουμε από την ουρά μας την σελίδα που βρίσκεται στην αρχή και βάζουμε μέσα την καινούργια σελίδα που έχουμε ζητήσει.

3.2.7.1 Ο συνολικός κώδικας του αρχείου lru.h

```
#ifndef LRU_H
#define LRU_H
#pragma once

#include <iostream>
#include <queue>
#include <string>
#include "../inverted_page_table.h"
#include "../trace.h"
#include "../stats.h"


using namespace std;

void lru(queue<Trace> traces,unsigned number_of_frames){
    InvertedPageTable IPT(number_of_frames);
    queue<Trace> lru_queue;

    int hitcount = 0;
    int pagefaultcount = 0;
```

```
    while (!traces.empty()) {
        Trace trace = traces.front();
        traces.pop();
        cout << trace.a << ' ';
        if(IPT.search(trace)){

            hit();
            hitcount++;

            queue<Trace> t = lru_queue;
            queue<Trace> newQ;
            while (!t.empty())
            {
                if (t.front().virtual_page_number !=
trace.virtual_page_number){
                    newQ.push(t.front());
                }
                t.pop();
            }
            newQ.push(trace);

            lru_queue = newQ;
        } else {

            page_fault();
            pagefaultcount++;

            if(lru_queue.size() < number_of_frames){
                IPT.load_to_ram(trace);
                lru_queue.push(trace);
            } else {
                IPT.unload_from_ram(lru_queue.front());
                lru_queue.pop();
                IPT.load_to_ram(trace);
                lru_queue.push(trace);
            }
        }
    }
    cout << "Hits: " << hitcount << endl;
    cout << "PageFaults: " << pagefaultcount << endl;
}

#endif // LRU_H
```

3.2.8 Το αρχείο opt.h

Στον αλγόριθμο Optimal (OPT) αντικαθιστούμε τη σελίδα που δεν θα χρησιμοποιηθεί για το μεγαλύτερο χρονικό διάστημα στο μέλλον.

Σκοπός της Συνάρτησης:

Στην συνάρτηση predict() ψάχνουμε σε όλη την ουρά και βρίσκουμε ποιο από το trace είναι το πιο μακρινό (δηλαδή ποια σελίδα θα χρησιμοποιήσουμε πιο αργά στο μέλλον) σε σχέση με το trace που ζητάμε εμείς εκείνη την στιγμή .

```
Int predict(queue<Trace> traces, vector<Trace> opt_queue
```

Η συνάρτηση opt παίρνει δυο παραμέτρους:

- queue<Trace>traces: μια ουρά από traces.
- number_of_frames: τον αριθμό των frames που θα το ορίσει ο χρήστης.

```
void opt(queue<Trace> traces, unsigned number_of_frames)
```

Όταν έχουμε Hit() :

- Απλώς διαβάζουμε την σελίδα που ήδη υπάρχει μέσα στην μνήμη μας.

Όταν έχουμε PageFault() κάνουμε δυο ελέγχους :

- Αν έχουμε χώρο στην ουρά, τότε βάζουμε την σελίδα μέσα.

➢ Αν η ουρά είναι γεμάτη, τότε καλούμε την συνάρτηση predict(), που μας επιστρέφει ένα trace και το οποίο διαγράφουμε από την ουρά. Στη συνέχεια, βάζουμε μέσα την καινούργια σελίδα που έχουμε ζητήσει.

3.2.8.1 Ο συνολικός κώδικας του αρχείου opt.h

```
#ifndef OPT_H
#define OPT_H
#pragma once

#include <iostream>
#include <queue>
#include <string>
#include "../inverted_page_table.h"
#include "../trace.h"
#include "../stats.h"

using namespace std;

int predict(queue<Trace> traces, vector<Trace> opt_queue){

    int result = -1, farthest = -1;

    for (int i = 0; i < opt_queue.size(); i++) {

        int j = 0;
        queue<Trace> dublicateQ = traces;
        while (!dublicateQ.empty()) {
            Trace trace = dublicateQ.front();
            dublicateQ.pop();

            if (opt_queue[i].virtual_page_number ==
trace.virtual_page_number){
                if (j > farthest) {
                    farthest = j;
                    result = i;
                }
                break;
            }
            j++;
        }
```

```
        if (j == traces.size())
            return i;
    }
    return (result == -1) ? 0 : result;
}

void opt(queue<Trace> traces,unsigned number_of_frames) {


    InvertedPageTable IPT(number_of_frames);
    vector<Trace> opt_queue;

    int hitcount = 0;
    int pagefaultcount = 0;

    while (!traces.empty()) {

        Trace trace = traces.front();
        traces.pop();

        cout << trace.a << ' ';

        if(IPT.search(trace)){
            hit();
            hitcount++;
        } else {
            page_fault();
            pagefaultcount++;
            if(opt_queue.size() < number_of_frames){
                IPT.load_to_ram(trace);
                opt_queue.push_back(trace);

            } else {
                //predict
                int position = predict(traces,opt_queue);
                IPT.unload_from_ram(opt_queue[position]);
                opt_queue.erase(opt_queue.begin()+position);
                IPT.load_to_ram(trace);
                opt_queue.push_back(trace);
            }
        }
    }
    cout << "Hits: " << hitcount << endl;
    cout << "PageFaults: " << pagefaultcount << endl;
}

#endif
```

3.3 Τα αποτελέσματα των αλγορίθμων

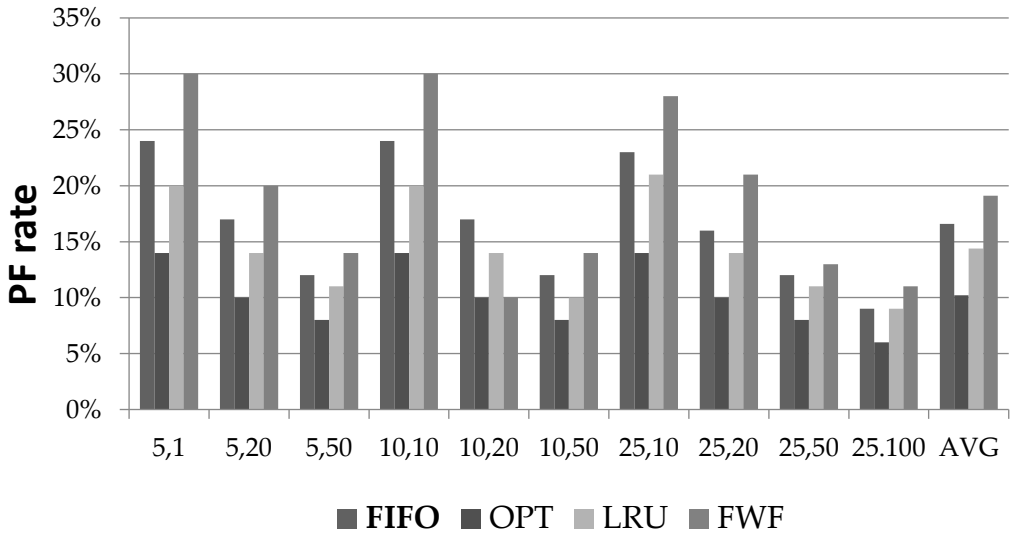
Έχουμε βάλει το πρόγραμμα να δέχεται τα εξής ορίσματα από τον χρήστη:

- Τον αλγόριθμο αντικατάστασης.
- Τον αριθμό πλαισίων της κεντρικής μνήμης (number of frames - f).
- Το πλήθος των σελίδων που θα διαβάζει εναλλάξ το πρόγραμμα από το κάθε αρχείο ίχνους, ώστε στην κεντρική μνήμη να υπάρχουν σελίδες και από τις 2 διεργασίες (quantum - q).

Στα πλαίσια της πτυχιακής αυτής πάρθηκαν, συνολικά, μετρήσεις από τους 4 αλγορίθμους και 10 εναλλακτικές παραμέτρους:

1. πείραμα 1: (q = 5, number_of_frames = 10)
2. πείραμα 2: (q = 5, number_of_frames = 20)
3. πείραμα 3: (q = 5, number_of_frames = 50)
4. πείραμα 4: (q = 10, number_of_frames = 10)
5. πείραμα 5: (q = 10, number_of_frames = 20)
6. πείραμα 6: (q = 10, number_of_frames = 50)
7. πείραμα 7: (q = 25, number_of_frames = 10)
8. πείραμα 8: (q = 25, number_of_frames = 20)
9. πείραμα 9: (q = 25, number_of_frames = 50)
10. πείραμα 10: (q = 25, number_of_frames = 100)

Στην εικόνα που ακολουθεί εμφανίζονται τα ποσοστά σφάλματος σελίδας (Page Fault Rate – PF rate), σε κάθε πείραμα, καθώς και ο μέσος όρος για κάθε αλγόριθμο αντικατάστασης. Οι τιμές στον οριζόντιο άξονα υποδεικνύουν τις τιμές των 2 παραμέτρων (q, frames) και στον κάθετο το PF rate.



Εικόνα 16. Συγκριτικά Αποτελέσματα των 4 αλγορίθμων αντικατάστασης

3.4 Συμπεράσματα

Με βάση τα αποτελέσματα (τα οποία επαληθεύουν τη βιβλιογραφία), καταλαβαίνουμε τα εξής:

- Ο αλγόριθμος FIFO δεν είναι κάλος γιατί παρόλο που είναι απλός στην υλοποίηση του, η λογική στην οποία βασίζεται δεν είναι συχνά σωστή και έτσι έχει κακή απόδοση.

  Χαρακτηριστικά:

  - Κακή Απόδοση
  - Εύκολη υλοποίηση

- Ο αλγόριθμος LRU έχει καλές επιδόσεις (καλύτερες από τον FIFO και από τον FWF), όμως είναι σχετικά ακριβός στην υλοποίηση του γιατί χρειάζεται να καταγράφεται ο χρόνος που έγινε αναφορά σε μία σελίδα κάθε φορά που χρησιμοποιείτε για ανάγνωση ή τροποποίηση, κάτι που έχει μεγάλο κόστος.

  Χαρακτηριστικά:

  - Μέτρια Απόδοση
  - Μέτρια υλοποίηση

- Ο αλγόριθμος OPT έχει πάντα καλυτέρα αποτελέσματα από τους FIFO και LRU, το πρόβλημα όμως είναι ότι δεν μπορούμε να ξέρουμε πάντα τη μελλοντική συμπεριφορά του προγράμματος. Επομένως, δεν μπορεί να υλοποιηθεί στα προγράμματα αλλά μπορεί να χρησιμεύσει σαν μέσο σύγκρισης για τους υπολοίπους αλγορίθμους.

  Χαρακτηριστικά:

  - Βέλτιστη Απόδοση
  - Όχι πάντα εφαρμόσιμος (πρέπει να γνωρίζουμε το μέλλον)

- Ο αλγόριθμος FWF είναι χειρότερος από την LRU, διότι επιφέρει αυστηρά μεγαλύτερο αριθμό σφαλμάτων στις περισσότερες εισόδους.

Χαρακτηριστικά:

- Μέτρια Απόδοση
- Εύκολη υλοποίηση

ΠΙΝΑΚΑΣ ΕΙΚΟΝΩΝ

Βιβλιογραφία

1. Andrew S. Tanenbaum, “Σύγχρονα Λειτουγικά Συστήματα”, 3η αμερικανική έκδοση.
2. Κ.Γ. Μαργαρίτης, “Διαχείριση μνήμης”, Σημειώσεις Λειτουργικών Συστημάτων, Τμήμα Εφαρμοσμένης Πληροφορικής, Πανεπιστήμιο Μακεδονίας, http://www.it.uom.gr/teaching/moss/mnimi-2.pdf.
3. Διομήδης Σπινέλλης, “Κατακερματισμός”, Τμήμα Διοικητικής Επιστήμης και Τεχνολογίας Οικονομικό Πανεπιστήμιο Αθηνών, https://www2.dmst.aueb.gr/dds/c2/hash/indexw.htm.
4. Behrouz Forouzan, “Εισαγωγή στην Επιστήμη των Υπολογιστών”, http://software.hpclab.ceid.upatras.gr/opsys/opsys/Lecture/chapter3_mem.pdf
5. Nectarios Koziris, “Εικονική Μνήμη”, National Technical University of Athens (NTUA), CSLab**,** http://www.cslab.ece.ntua.gr/courses/comparch/2010/files/fall2010_11/chapter7-ca-Memory_Hierarchy-partC-virtual_memory-Fall_2010.pdf
6. Γεώργιος Ξυλωμένος, “Διαχείριση Μνήμης”, Σημειώσεις Λειτουργικών Συστημάτων, Τμήμα Πληροφορικής, Οικονομικό Πανεπιστήμιο Αθηνών,https://eclass.aueb.gr/modules/document/file.php/03_Operating_Systems.pdf
7. Amir Kamil, “Inverted Page Tables”, UC Berkeley, https://web.eecs.umich.edu/~akamil/teaching/sp04/040104.pdf
8. Wikipedia, “Page Table”, https://en.wikipedia.org/wiki/Page_table
9. Junfeng Yang, “Operating Systems”, https://www.cs.columbia.edu/~junfeng/10sp-w4118/lectures/l21-page.pdf
10. Neeraj Mishra, “Page Replacement Algorithm”, https://www.thecrazyprogrammer.com/
11. Ιωάννης Καραγιάννης , “Αλγόριθμοι ΄Αμεσης Απόκρισης” , https://www.ceid.upatras.gr/webpages/faculty/caragian/courses/ln-09.pdf