ISLAB HACK: Βασικές Έννοιες της Αρχιτεκτονικής INTEL - IA32 για LINUX & WINDOWS

Αθήνα 2003

Περιεχόμενα

Περιήγηση στην αρχιτεκτονική IA32 και των λειτουργικών συστημάτων Linux και Ms-Windows

1 Γενικά

Το κείμενο αυτό γράφτηκε στα πλαίσια της πτυχιακής μου εργασίας που αφορά τις επιθέσει που βασίζονται σε αδυναμίες υπερχείλισης της μνήμης. Ο σκοπός αυτού του κειμένου είναι να κάνει μια περιήγηση στην **32bit αρχιτεκτονική** της γνωστότερης οικογένείας των επεξεργαστών **της Intel** καθώς και της αρχιτεκτονικής των δύο γνωστότερων **λειτουργικών συστημάτων** που βασίζονται κυρίως στην τεχνολογία της Intel και είναι το **Linux** και τα **Ms-Windows**. Αυτό το κείμενο μπορεί να χρησιμοποιηθεί σαν κείμενο που θα αυξήσεις την θεωρητική υποδομή κάποιου που θέλει να έχει ένα γενικότερο υπόβαθρο ώστε να μπορεί να καταλαβαίνει πως ακριβώς ένας **κακόβουλος χρήστης(Blackhat hacker)** εκμεταλλεύεται αυτή την αρχιτεκτονική με τελικό σκοπό να καταλάβει(Compromise) ένα υπολογιστικό σύστημα που δεν του ανήκει.

Επειδή αυτό το κείμενο γράφτηκε με κύριο σκοπό να μπορέσει να γίνει κατανοητό το πώς γίνονται οι επιθέσει που βασίζονται σε **Buffer Overflow** αδυναμίες σε πολλά σημεία θα επισημαίνονται ποίες ακριβώς είναι οι γνώσει που χρησιμοποιεί ένας κακόβουλος χρήστης για να μπορέσει να χτίσει ένα **Buffer Overflow Exploit** (πρόγραμμα που εκμεταλλεύεται την αδυναμία υπερχείλισης της μνήμης).

2 Αρχιτεκτονική της Intel

Η ιστορία της αρχιτεκτονικής των επεξεργαστών της Intel ξεκινάει με από τον επεξεργαστή 8086 μέχρι και τον σημερινό(2003) Pentium 4 και τον XEON. Η βασική λειτουργία του επεξεργαστή από τον 8086 μέχρι τον Pentium έχει αλλάξει δραματικά όμως δυο είναι τα βασικότερα χαρακτηριστικά που άλλαξαν μέχρι σήμερα. Φυσικά αναμένονται να γίνουν αρκετές αλλαγές με τον πρωτοεμφανιζόμενο επεξεργαστή των 64bit της Intel με το όνομα **Itanium** όμως η 32bit τεχνολογία θα συνεχίσει να επικρατεί για πολλά χρόνια ακόμα.

Η βασικότερη αλλαγή στην εξέλιξης της τεχνολογίας της Intel ήταν αφενός η μετάβαση από την 16bit στη **αμιγώς 32bit** τεχνολογία αλλά και η ριζική αλλαγή στο **Memory Addressing**. Ο μεγαλύτερος σταθμός στην ιστορία της Intel μέχρι σήμερα ήταν ο επεξεργαστής **i386**. Αυτό συμβαίνει όχι μόνο γιατί με τον επεξεργαστή εισάγεται για πρώτη φορά η 32bit τεχνολογία αλλά πρωτοεμφανίζεται η **δυνατότητα διαχείρισης πολύ μεγάλης ποσότητας μνήμης (4GB)** που απαιτούνται από λειτουργικά συστήματα υψηλών επιδόσεων όπως το UNIX.

Ο επεξεργαστής αυτός ήταν τόσο σημαντικός γιατί με αυτόν εμφανίστηκε για πρώτη φορά στην ιστορία των PC η δυνατότητα για να εφαρμοστεί το multitasking, το Paging και η δυνατότητα για multi-users που εφαρμοζόταν μέχρι τότε μόνο σε πολύ ακριβά mainframe συστήματα. Το σημαντικότερο όμως από όλα δεν ήταν ότι εμφανίστηκε ένας τέτοιος επεξεργαστής αλλά κυρίως ότι με πολύ έξυπνο τρόπο μπόρεσε να κρατηθεί η συμβατότητα με του περασμένους επεξεργαστές της Intel κατά συνέπεια και στο λογισμικό που είχε αναπτυχθεί τόσα χρόνια για τα PC.

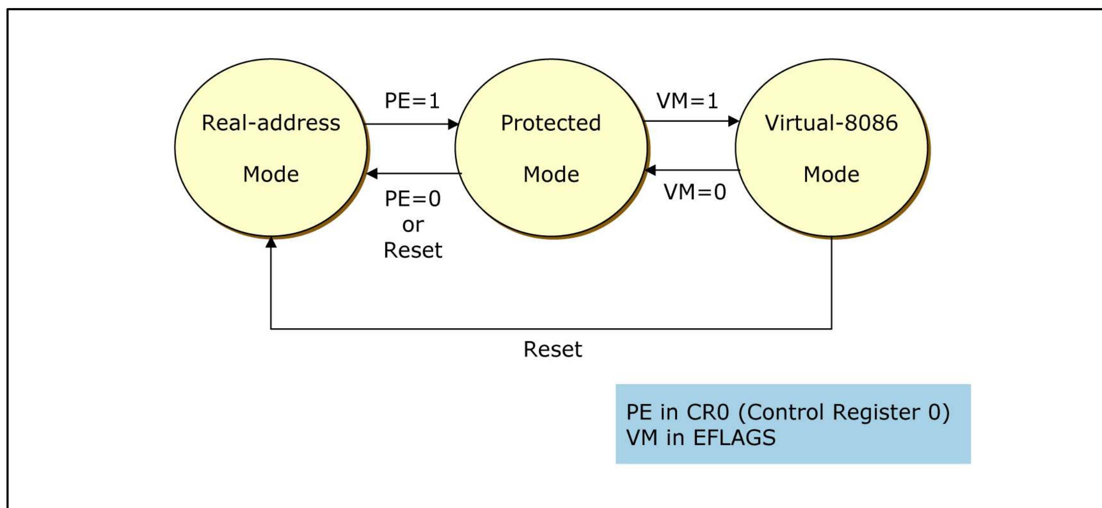
Ακόμα και σήμερα στον Pentium 4 εξακολουθεί αυτή η τεχνολογία που εμφανίσθηκε πρώτη φορά στον 80386 να χρησιμοποιείται σήμερα ακριβώς όπως λειτουργούσε και τότε. Οι βασικές βελτιώσεις που έχουν γίνει από την εμφάνιση του 386 μέχρι σήμερα στον Pentium 4 είναι στην μετάβαση από την τεχνολογία CISC στην RISC καθώς και η δραματική αύξησης της δυνατότητας των επεξεργαστών να εκτελούν παράλληλα τις **εντολές επεξεργαστή (Opcode)**. Ακόμα προστέθηκε η δυνατότητα υπολογισμού μεγάλων διανυσμάτων με την τεχνολογία MMX που βοήθησε περισσότερο τα γραφικά στους υπολογιστές. Το βασικό χαρακτηριστικό που διαχωρίζει την προ-32bit εποχή με αυτή είναι κυρίως το Memory Addressing. Αν και στην φιλοσοφία του που θα περιγράψουμε και παρακάτω, το Memory Addressing, παραμένει ακριβώς το ίδιο έχουν γίνει μερικές βελτιώσει από τη εποχή του 386 μέχρι τον σημερινό Pentium 4 που στην πράξη είναι διαφανείς για τις εφαρμογές. Οι βελτιώσεις αυτές είναι η **cach memory επιπέδου(level) 2** που κύριο σκοπό έχει να κρατά τις

Linear Address που είδη έχει υπολογίσει ο επεξεργαστής ώστε να μην χρειάζεται να τις ξανάυπολογιζει.

Το πόσο σημαντική ήταν αυτή η αλλαγή που έφερε ο i386 φαίνεται γιατί ακόμα και σήμερα ο πυρήνας των λειτουργικών συστημάτων όπως είναι τα MS-windows(NT generation) ή το Linux που χρησιμοποιείται όταν το εκάστοτε λειτουργικό θα εγκατασταθεί σε **intel** επεξεργαστή, χαρακτηρίζεται σαν **«Πυρήνας αρχιτεκτονικής i386»**. Παρακάτω παρουσιάζεται η αρχιτεκτονική **IA32.**

2.1 Intel IA32

Για να μπορέσει η Intel να κρατήσει συμβατότητα με την 16bit τεχνολογία της χρησιμοποίησε ένα πολύ έξυπνο μηχανισμό. Κάθε επεξεργαστής που ακολουθεί την **Intel IA32** θα πρέπει να υποστηρίζει 3 mode λειτουργίας το **Real-Mode** το **Protected-Mode** και το **Virtual-8086 mode**. Οι **καταστάσεις(modes)** που μπορεί να βρεθεί, όπως αναφέρθηκε προηγουμένως, ένας Intel 32bit επεξεργαστής μέχρι και σήμερα στον Pentium 4 φαίνονται στο παρακάτω σχήμα.



Σχήμα 1

Η χρησιμότητα του κάθε mode είναι η εξής:

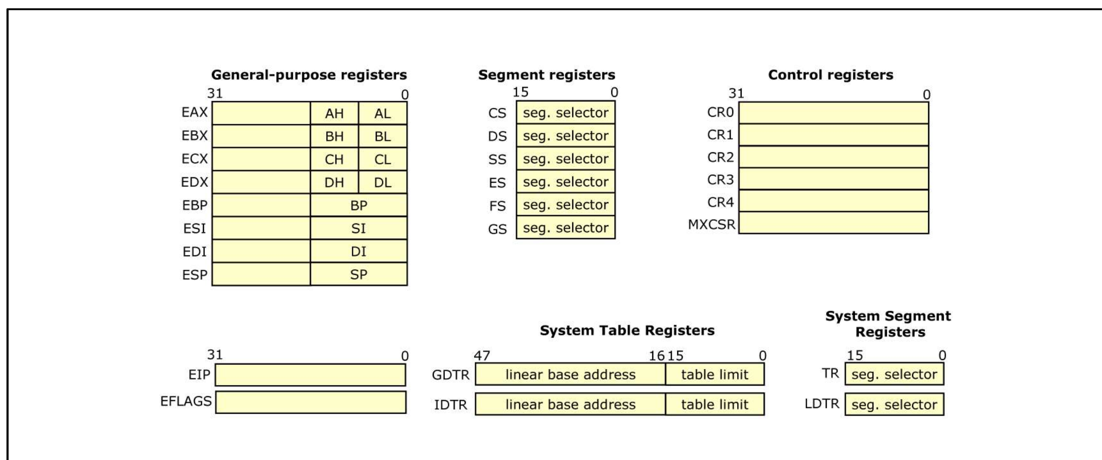
- **Real-address Mode :** Σε αυτή την κατάσταση ο επεξεργαστής θα βρεθεί όταν ανοίξει για πρώτη φορά ο υπολογιστής δηλαδή πριν φορτωθεί το λειτουργικό σύστημα. Σε αυτό το mode το **Memory Addressing** είναι αυτό που χρησιμοποιείται στον 8086 που αυτό στην πράξη σημαίνει 3 πράγματα:
    - Στον επεξεργαστή αυτό μπορεί να φορτωθεί λειτουργικό σύστημα MS-DOS ακόμα και αν αυτός είναι Pentium 4. Αυτό δεν θα μπορούσε να συμβεί αν δεν υπήρχε το Real-Mode γιατί το DOS είναι μια **αμιγώς 16bit εφαρμογή**.
    - Οι καταχωριστές CS, DS, SS χρησιμοποιούνται ως καταχωριστές βάσης και όχι σαν Selectors(index to memory Base).
    - Μπορεί να γίνει allocation μόνο 1 MB μνήμης από τον επεξεργαστή και για αυτό παλαιοτέρα είχε εισαχθεί το **Extended Memory Model** ώστε να μπορεί το σύστημα να διαχειριστεί περισσότερη από 1MB μνήμης.

- **Protected Mode:** Η κατάσταση αυτή του επεξεργαστή ενεργοποιείται μόνο από το λειτουργικό σύστημα αν αυτό απαιτείται. Όλα τα σημερινά λειτουργικά συστήματα Windows NT/2000/XP και Linux «τρέχουν» σε ένα **αμιγώς 32bit Protected Mode**. Λειτουργικά συστήματα όπως Windows 3.x/95 καθώς και τα Windows 98/Me «τρέχουν» σε **«υβριδικό mode»** δηλαδή αναλόγως με τις ανάγκες των εφαρμογών τρέχουν σε **Real** ή **σε Protected mode.** Η δυνατότητες που έχουν τα 32 bit συστήματα δεν θα μπορούσαν να υλοποιηθούν αν ο επεξεργαστής με το Protected mode δεν τις υποστήριζε. Οι βασικότερες από τις δυνατότητες αυτές είναι:
    - **Multitasking** που στηρίζεται στον **Memory Allocation / Addressing μηχανισμό** του επεξεργαστή.

   - **Flat Memory Addressing** που βασίζεται στην δυνατότητα του επεξεργαστή να χειρίζεται Segments των **4GB** αφήνοντας την δυνατότητα του Segmentation μόνο για ειδικές περιπτώσεις. Έτσι αυξάνονται σημαντικά οι επιδώσεις των εφαρμογών.

- **Virtual – 8086 Mode:** Ο λόγος που αυτό το mode υλοποιήθηκες αλλά και χρησιμοποιήθηκες ήταν για να δοθεί η δυνατότητα στα λειτουργικά συστήματα να τρέχουν εφαρμογές που έπρεπε να τρέξουν οπωσδήποτε σε Real-mode αλλά το λειτουργικό σύστημα έπρεπε και αυτό με την σειρά του να τρέχει σε Protected mode. Αυτό το mode χρησιμοποιεί οποιοδήποτε λειτουργικό σύστημα προσπαθεί να τρέξει μία εφαρμογή που έχει υλοποιηθεί σε 16bit Real-mode ενώ αυτό τρέχει σε Protected mode όπως για παράδειγμα τα Windows 98.

Για να μπορέσουν να υποστηριχθούν τα παραπάνω modes πρέπει όλοι οι επεξεργαστές της Intel μέχρι και τον Pentium 4 να έχουν **τουλάχιστον** τους καταχωριστές του *σχήματος 2.2*. Το σχήμα αυτό δείχνει την εσωτερική δομή των καταχωριτών ενός 32bit Intel επεξεργαστή. Οι καταχωριτές αυτοί είναι οι ελάχιστοι δυνατή γιατί αν χρειάζεται να υποστηριχθούν άλλες τεχνολογίες στον επεξεργαστή ενδεχομένως να χρειάζονται και άλλοι καταχωριτές.



Σχήμα 2

Η χρησιμότητα των καταχωριστών αυτών θα φανεί καλύτερα παρακάτω που θα παρουσιαστούν τα Memory Addressing που υποστηρίζουν τα παραπάνω mode ενός 32bit επεξεργαστή. Ενδεικτικά εδώ θα αναφερθεί ότι οι Segment registers αναλόγως το mode έχουν διαφορετικό λόγο ύπαρξης. Ο **CR0 register** είναι αυτό που **χρησιμοποιείται για να γίνεται μετάβαση** από **Real-mode** σε **Protected-mode**. Οι System Table Registers χρησιμοποιούνται από τον επεξεργαστή για το Segmentation ενώ ο καταχωριστής **CR3** χρησιμοποιείται συνήθως για το Paging. Παρακάτω θα γίνει μια περιήγηση στο memory addressing των δύο βασικών mode ενός επεξεργαστή **Intel IA32**.

2.2 Το memory addressing Real Mode

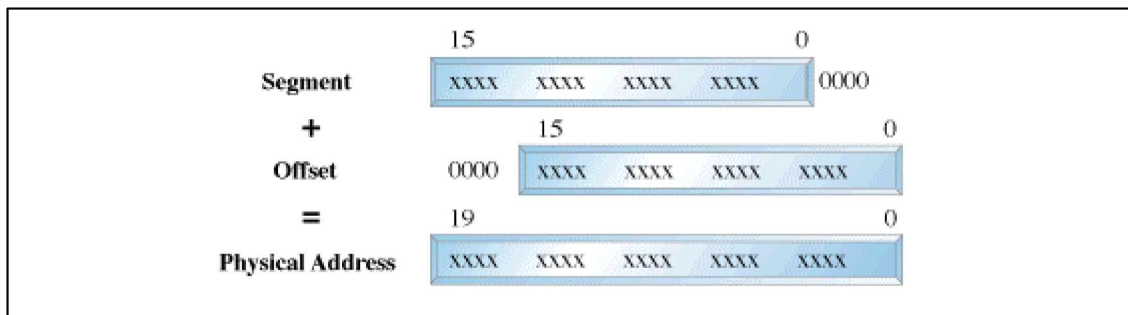
Ο λόγος που περιγράφεται αυτό το Mode δεν είναι γιατί χρειάζεται να το γνωρίζει κανείς ώστε να αναπτύξει εφαρμογές ή να γράψει κάποιο **Exploit**(κακόβουλο πρόγραμμα) για ένα Intel επεξεργαστή σήμερα εφόσον αυτό το mode δεν χρησιμοποιείται παρά μόνο σε ειδικές περιπτώσεις. Ο λόγος που θα γίνει αυτό είναι γιατί κατανοώντας το memory addressing της 16bit τεχνολογίας γίνεται ευκολότερα κατανοητό το πώς λειτουργεί αυτό της 32 bit τεχνολογίας. Τέλος περιγράφοντας με αυτή την σειρά τους δύο μηχανισμού memory addressing που υποστηρίζει ένας Intel IA-32 επεξεργαστής μπορεί να γίνει κατανοητό πώς έγινε η μετάβαση από την μια τεχνολογία στην αλλά και πώς διατηρήθηκε η συμβατότητα.

Στο **Real mode** όπως είπαμε ο επεξεργαστής διαχειρίζεται την μνήμη όπως ένας 8086 ακόμα και αν αυτός είναι ο Pentium 4. Αυτό σημαίνει ότι μπορεί να χειριστεί μόνο 1 MB φυσικής μνήμης. Αυτό οφείλεται στο γεγονός ότι για της διευθύνσεις της μνήμης χρησιμοποιούνται μόνο **20bit** (2^{20} = 1 MB). Όπως όμως και στα σημερινά λειτουργικά συστήματα έτσι και τότε

υπήρχαν δύο βασικές ανάγκες. Η πρώτη ανάγκη ήταν ότι κάθε πρόγραμμα θα έπρεπε να «τρέχει» σε ένα δικό του χώρο άρα η μνήμη θα έπρεπε να χωρίζεται σε **Segments**(τμήματα). Η δεύτερη ανάγκη ήταν ότι το λειτουργικό σύστημα θα έπρεπε να είναι αυτό που θα αποφασίζει που θα ξεκινάει το segment μιας εφαρμογής μέσα στην μνήμη. Για το λόγο αυτό κάθε πρόγραμμα αποφασιστικέ ότι θα αναφέρεται σε λογικές διευθύνσεις ενώ με την βοήθεια του λειτουργικού συστήματος θα παραγόταν από τον επεξεργαστή η φυσική διεύθυνση των 20 bit.

Για τους λόγου που περιγράφονται παραπάνω μέσα στον 16 bit επεξεργαστή υπήρχαν 3 καταχωριστές **μεγέθους 16bit** που υπάρχουν ακριβώς οι ίδιοι και στους 32 bit σήμερα. Αυτοί οι καταχωριστές είναι οι **CS, DS, SS** και ο σκοπός τους αρχικά ήταν να δείχνουν την διεύθυνση που ξεκινούν τα 3 Segment που δεσμεύονται για κάθε εφαρμογή. Αυτά τα Segment που κατά σειρά αντιστοιχούν σε αυτούς τους τρεις καταχωριστές είναι το **Code Segment** το **Data Segment** και το **Stuck Segment**. Ο σκοπός των καταχωριτών αυτών είναι ακριβώς ο ίδιος σε ένα 32bit επεξεργαστή αλλά *μόνο όταν αυτός βρίσκεται* σε Real-mode.

Όπως είπαμε όμως ένα πρόγραμμα όταν αναφέρεται στην μνήμη στην πράξη δεν αναφέρεται σε κάποια **φυσική διεύθυνση μνήμης** αλλά στην **απόσταση(Offset)** που έχει η διεύθυνση αυτή από την αρχή του Segment. Για να μπορέσει ο επεξεργαστής να δώσει την φυσική διεύθυνση των 20bit όταν ένα πρόγραμμα του το «ζητήσει» πρέπει να προσθέσει την τιμή του **Offset** του προγράμματος με τον **κατάλληλο Segment Register**. Η τιμή του segment register έχει προκαθοριστεί από το λειτουργικό σύστημα ποία θα είναι. Επειδή όμως και οι δύο τιμές θα είναι **16bit** και η πρόσθεση τους πρέπει να παράγει μία φυσική διεύθυνση 20 bit θα χριστή το **Offset** να πολλαπλασιαστή με 16 και σε δεύτερη φάση να γίνει η πρόσθεση. Στο παρακάτω σχήμα παρουσιάζεται το πώς παράγεται η **φυσική διεύθυνση μνήμης σε ένα 16bit σύστημα.**



Σχήμα 3

Τα προβλήματα που παρουσιάστηκαν πολλή σύντομα λόγο αυτού του μηχανισμού memory Addressing ήταν πολλά. Για παράδειγμα λόγο των 16 bit μόνο που είχε κάθε *Segment Register* κάθε Segment δεν μπορούσε να είναι μεγαλύτερο των **64Kbyte**. Για να ξεπεραστούν τέτοια προβλήματα είχαν χρησιμοποιηθεί διάφορα **μοντέλα προγραμματισμού**, και όσοι έχουν προγραμματίσει σε Assembly σίγουρα τα γνωρίζουν. Τα μοντέλα αυτά ήταν το **tiny, small, medium, compact, large** και το **huge**. Εκτός από αυτά τα μοντέλα που στην πράξη τα χρησιμοποιούσαν οι προγραμματιστές ακόμα και μέχρι το 1993 υπήρχαν και άλλα μοντέλα που εμφανίστηκαν τότε για να μπορέσει να ξεπεραστεί το όριο του **1 MB**. Η ανάγκη που προέκυψε τότε ήταν ότι οι εφαρμογές είχαν αρχίσει είδη να χρίζονται περισσότερα από 65K η κάθε μία τότε εμφανίστηκε το μοντέλο **Extended Memory model** που στο MS-DOS το υλοποιούσε η διεργασία **Himem.exe**.

Τα προβλήματα που είχαν να αντιμετωπίσουν οι προγραμματιστές που ανέπτυσσαν τους Compilers, τους linkers και τα λειτουργικά συστήματα ήταν ότι έπρεπε να δίνουν τις κατάλληλες τιμές στους καταχωριστές Segment registers για να δουλέψει σωστά ένα πρόγραμμα υψηλών απαιτήσεων για την εποχή. Ευτυχώς τα προβλήματα αυτά ήρθε να τα λύσει το **32bit Protected mode**. Φυσικά αυτά θα είναι και τα προβλήματα που θα πρέπει να αντιμετωπιστούν αν αντί για το 32bit Protected mode χρησιμοποιηθεί το Real-mode ακόμα και αν ο επεξεργαστής αυτός είναι ο Pentium 4.

2.3 To memory addressing 32bit Protected Mode

Με την εμφάνιση του 80386 για πρώτη φορά στα PC εμφανίζετε ένας αμιγώς 32 bit επεξεργαστής. Ένα βασικό πλεονέκτημα ήταν το γεγονός ότι μια 32bit εντολή χρειαζόταν τον ίδιο χρόνο για να εκτελεστεί όσο μια 16 bit εντολή ή όσο μια 8bit εντολή. Αυτό έχει σαν αποτέλεσμα οι εντολές που ένας Compiler μεταφράζει σε δύο opcoede ενός 16 bit επεξεργαστή να μεταφράζεται μόνο σε ένα opcode όταν ο επεξεργαστής είναι 32bit. Συνεπώς ένα πρόγραμμα μειώνεται σε μέγεθος άρα η ταχύτητα εκτέλεσης ενός προγράμματος μπορεί στην πράξη να διπλασιαστεί. Ένα δεύτερο χαρακτηριστικό που η ιστορία απέδειξε ότι είναι το σημαντικότερο για να επικρατήσει μια τεχνολογία ήταν ότι προγράμματα προηγούμενης τεχνολογίας μπορούσαν να τρέξουν χωρίς ιδιαίτερα προβλήματα στον νέο επεξεργαστή κάτι που παρακινούσε και συνεχίζει να παρακινεί τους προγραμματιστές και χρήστες να υποστηρίζουν την τεχνολογία αυτή.

Το σημαντικότερο όμως χαρακτηριστικό που εισάγει για πρώτη φορά ο 386 είναι μια υποδομή για την διαχείρισης της μνήμης που μέχρι και σήμερα χρησιμοποιείται απαράλλαχτη. Αυτή η υποδομή είναι η **memory management unit (MMU)** που εισάγει ένα νέο **memory addressing mode** πάνω από το είδη υπάρχον **Real-mode** και αυτό είναι το **Protected mode**.

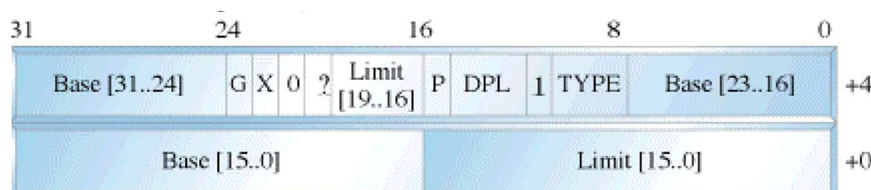
Το **Protected mode** από την εποχή του 386 μέχρι σήμερα στον Pentium 4 προσφέρεται ένα πολύ μεγάλο βαθμό ευελιξίας κάνοντας δυνατή την διαχείριση πολύ μεγάλων Segment μνήμης μεγέθους των **4GB** διαθέτοντα έτσι ένα **ενιαίο (Flat) χώρο μνήμης (address space)** σε ένα πρόγραμμα. Επίσης εκτός από την φυσική διεύθυνση των **4GB** σε ένα Segment με το αυτό το Memory Addressing ο επεξεργαστής μπορεί να διαχειριστεί **64TB** (64,000,000,000,000 bytes) **virtual memory**. Τέλος με αυτό το memory addressing προστίθεται ένας βαθμός προστασίας με σκοπό να υποστηριχθούν συστήματα, όπως το UNIX, που είδη είχαν την απαίτηση για ασφάλεια αρκετά χρόνια πριν από την εμφάνιση του **i386**.

Στο Protected mode όπως αναφέρεται παραπάνω οι Segment registers, όπως ο CS, δεν χρησιμοποιούνται για αν κρατούν την **διεύθυνση βάσης** όπως συμβαίνει στο Real Mode. Οι καταχωριστές αυτή παίρνουν την ιδιότητα του **Selector** δηλαδή χρησιμοποιούνται ως **δείκτες (index)** σε κάποιους πίνακες στην μνήμης που κάθε εγράφη τους περιέχει μία **32-bit διεύθυνση βάσης**. Με αυτό τον τρόπο προσθέτονται μία **32bit βάση με ένα 32bit Offset** απλοποιώντας δραματικά το προγραμματισμό των 32bit εφαρμογών σε σχέση με αυτές των 16bit.

Οι πίνακες που αναφέρονται παραπάνω είναι τρεις και ο κάθε ένας τους μπορεί να βρίσκεται στην ROM ή συνήθως στην RAM. Αυτοί οι πίνακες αρχικοποιούνται από το λειτουργικό σύστημα αλλά χρησιμοποιούνται από τον επεξεργαστή όπως συνέβαινε αντίστοιχα με τους Segment Registers στην 16bit τεχνολογία ή στο Real-mode. Οι πίνακες αυτοί είναι:

- **GDT (Global Descriptor Table)** : Ο πίνακας αυτός είναι ένας για ολόκληρο το σύστημα και είναι πάντα προσβάσιμος.
- **LDT (Local Descriptor Table)** : Ένας τέτοιος πίνακα χρησιμοποιείται για κάθε **Task** (βλέπε multitasking) όταν χρησιμοποιείται segmentation. Στην πράξη μπορεί να μην χρησιμοποιείται καθόλου ή μπορεί να χρησιμοποιείται μόνο ένα ή πολλοί τέτοιοι πίνακες σε ένα σύστημα. Τέλος **μόνο ένας** είναι ενεργός κάθε στιγμή.
- **IDT (Interrupt Descriptor Table)** : Οι πίνακες αυτοί χρησιμοποιείται για να τα **interrupts** του **συστήματος**.

Στο Σχ*ήμα 2.4* φαίνεται αναλυτικά μία εγράφη σε ένα πίνακα όπως οι τρεις παραπάνω. Η εγράφη αυτή λέγεται **Descriptor** γιατί εκτός από την Base Address περιέχει και άλλες πληροφορίες για ένα segment όπως για παράδειγμα το όριο του μεγέθους του Segment ή τα δικαιώματα χρήσης σε αυτό.



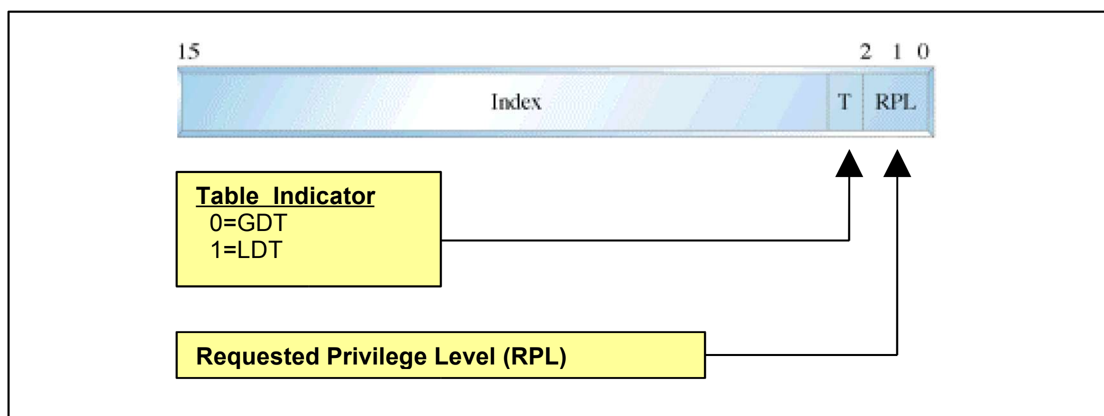
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Σχήμα 4

Όπως φαίνεται ένας **Descriptor** σε ένα πίνακα GDT, LDT ή IDT αποτελείται από **δύο 32bit λέξεις**. Αυτό συμβαίνει γιατί σε έναν Descriptor πρέπει να υπάρχουν πληροφορίες που αφορούν τα δικαιώματα πρόσβασης που μπορεί να έχει μια εφαρμογή σε ένα Segment μαζί την Base Address που χρειάζεται ο επεξεργαστής για να μπορέσει να την προσθέσει στο **Offset** ενός προγράμματος και να παράγει την πραγματική διεύθυνση. Όλα τα πεδία ενός **Descriptor** έχουν ιδιαίτερη σημασία τα σημαντικότερα όμως είναι :

- Η **base address** που αποτελείται από **32bit**.
- Το **limit** που αποτελείται από **20bit**. Όταν το **Granularity** bit είναι ενεργοποιημένο τότε το όριο ενός Segment είναι 4K αλλιώς είναι 1Byte.
- Από τα **Control bits** τα σημαντικότερα είναι:
    - **Granularity bit** που αλλάζει το μέγεθος του ορίου ενός **Segment**.
    - Το **DPL (Descriptor Privilege Level)** που καθορίζει τα δικαιώματα χρήσης ενός Segment.
    - Το **D/B (Default operation size)** που καθορίζει αν το Segment θα είναι 16bit ή 32bit.

Ό όπως είπαμε όμως για να οδηγηθεί ο επεξεργαστής να επιλέξει ένα **Descriptor** από έναν πίνακα GDT ή LDT θα πρέπει πρώτα να διάβαση την τιμή ενός **Selector**. Το ρόλο του Selector όπως είπαμε θα τον αναλάβουν οι Segment registers. Οι registers αυτοί είναι 16bit ακόμα και αν το σύστημα είναι σε κατάσταση **Protected mode**. Αν και στο Real mode χρησιμοποιούνται και τα 16bit για τον ίδιο σκοπό στο **Protected mode** τα τρία τελευταία χρησιμοποιούνται για διαφορετικό σκοπό όπως φαίνεται στο παρακάτω σχήμα.

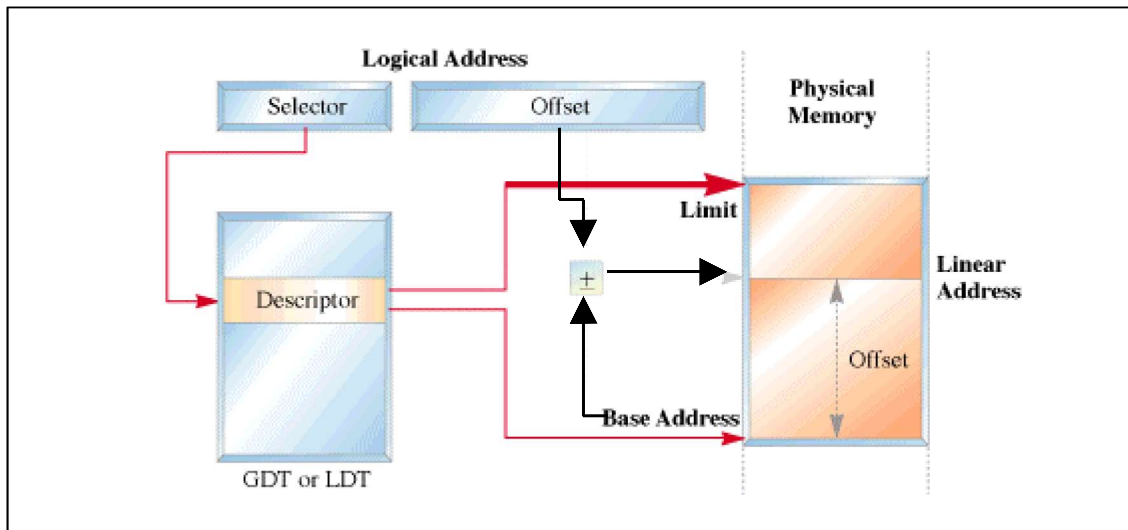


Σχήμα 5

Από το σχήμα φαίνεται ότι σε ένα **Selector** μόνο τα **13 bit** του χρησιμοποιούνται σαν **δείκτης (index)** σε ένα από του πίνακες GDT ή LDT. Ο τύπος του πίνακα καθορίζεται από τον **ένδικη T(table indicator)** ενώ τα δικαιώματα χρήσης με τα οποία ζητείται να γίνει πρόσβαση στο Segment υποδεικνύονται από το **RPL**.

Από όλα τα παραπάνω φαίνεται ότι στο **Protected mode** η λογική για να παραχθεί μια φυσική διεύθυνση παραμένει ακριβώς η ίδια όπως στο Real-mode ή στον 8086. Η φυσική διεύθυνση παράγεται από τον συνδυασμό μία ενός **16 bit Selector** και ενός 16 ή **32 bit Offset**. Η βασική διαφορά όμως είναι ότι δεν χρησιμοποιείται σαν διεύθυνση βάσης η τιμή του Selector αλλά

αυτή του **Descriptor** που υποδεικνύεται από τον πρώτο. Ο Descriptor θα περιέχει μία **διεύθυνση βάσης** που είναι 32bit. Η φυσική διεύθυνση που θα παραχθεί από την πρόσθεση της **διεύθυνσης βάσης** με το **Offset** ενός προγράμματος θα είναι **32bit**. Ενώ αυτή η 32bit τιμή που παράγεται είναι η **φυσική διεύθυνση** την λέμε **Linear Address**. Το όνομα αυτό δόθηκε γιατί όταν χρησιμοποιείται ο μηχανισμός του Paging από το λειτουργικό σύστημα η διεύθυνση αυτή αντιστοιχίζεται σε μία άλλη, διαφανώς, από το λειτουργικό σύστημα. Έτσι με τον μηχανισμό του Paging από την Linear Address το λειτουργικό σύστημα θα παράγει μια νέα **φυσική διεύθυνση**. Το paging θα παρουσιαστεί αμέσως παρακάτω. Στο παρακάτω σχήμα παρουσιάζεται πως ακριβώς παράγεται η **Linear address**.



Σχήμα 6

Τα προβλήματα που λύθηκαν με το **32bit Protected mode** ήταν εξαιρετικά σημαντικά απλοποιώντας τον προγραμματισμό. Αυτό είχε σαν συνέπεια να δοθεί η Hardware υποδομή ώστε να σχεδιαστούν αξιόπιστα, ρωμαλέο και υψηλών επιδόσεων λειτουργικά συστήματα που με την σειρά τους δίνουν την δυνατότητα ανάπτυξης εφαρμογών αντίστοιχης ποιότητας. Τα σημαντικότερα πλεονεκτήματα και οι δυνατότητες που προσφέρει το **Protected mode** είναι:

- Η πρόσβαση της μνήμης σε ένα Segment μπορεί να γίνεται μόνο με το **Offset** του προγράμματος και δεν χρειάζεται επιπλέον υπολογισμοί.
- Επειδή κάθε Segment μπορεί έχει μέγεθος μέχρι και **4GB** δεν χρειάζεται να γίνεται συνεχώς αλλαγή της τιμής των Segment Registers ακόμα και αν η εφαρμογή που τρέχει είναι πολύ μεγάλη δηλαδή χειρίζεται πολύ μεγάλες δομές δεδομένων. Αυτό έχει σαν αποτέλεσμα να αυξάνονται δραματικά οι επιδώσεις στην εκτέλεση προγραμμάτων με υψηλές απαιτήσεις.
- Τα **Offset** σε ένα πρόγραμμα μπορούν να ξεκινούν από το **0** όποια και να είναι η **φυσική διεύθυνση** από την όποια το Segment ξεκινάει. Αυτή η δυνατότητα καθιστά το Debugging πολύ εύκολο. Επίσης μπορεί ένα Segment να βρεθεί σε οποιαδήποτε περιοχή της μνήμης χωρίς να χρειάζεται να αλλαχθούν τα Offset ενός προγράμματος κάτι που ήταν αναγκαία κακό στα 16-bit συστήματα.
- Ένα **Offset** δεν πρέπει να ξεπερνά την διεύθυνση που καθορίζεται από το **Limit** που υπάρχει μέσα στο Descriptor. Με αυτό τον τρόπο μπορεί να εξασφαλιστεί ότι θα προκληθεί ένα **exception** που θα συλλάβει το λειτουργικό σύστημα εμποδίζοντας έτσι την παραβίαση των ορίων στης περιοχής μνήμης που έχει δεσμευτεί για ένα πρόγραμμα από το ίδιο.
- Παραπάνω ειπώθηκε ότι μέσα στον Descriptor υπάρχουν αρκετά bit που χρησιμοποιούνται για την ασφάλεια του segment από μη επιθυμητή πρόσβαση. Αυτή **η δυνατότητα δεν υπάρχει στο σύστημα όταν αυτό τρέχει σε Real Mode**. Αυτό έχει σαν αποτέλεσμα για παράδειγμα να μην μπορεί ένα πρόγραμμα να αλλάξει τον κώδικα ενός αλλού προγράμματος. Αυτό μπορεί να γίνει θέτοντας το κατάλληλο bit στο descriptor σαν Read-Only όταν το segment αυτό περιέχει κώδικα.
- Όπως ειπώθηκε πολλές φορές συνήθως ένα Segment μπορεί να έχει μέγεθος **4GB** σε αντίθεση με το Real mode που το μέγεθος είναι ενός Segment μπορεί να είναι

64KB. Το μέγεθος του Segment προκύπτεί από τις τιμές που μπορεί να πάρει το **Base** και το **Limit** ενός Descriptor. Έτσι ένας **Descriptor** με τιμή για το **Base 0** και με **Limit 0xFFFFF (20 bit)** μπορεί να ορίσει χώρο **1MB**. Συνήθως όμως είναι ενεργοποιημένο το **Granularity bit** που αυξάνει το μέγεθος των μονάδων σε **4K** από αυτό προκύπτει ότι έχοντας θέσει στο Base και στο Limit τις προηγούμενες τιμές θα έχουμε Segment μεγέθους **4GB(4K x 1MB = 4GB)** και όχι 1MB μόνο. Σε ένα GDT ή LDT μπορεί όμως να υπάρχουν **8192 Descriptors** αφού ο **Selector** χρησιμοποιεί 13bit για index προς αυτούς τους πίνακες. Αν λοιπών αθροίσουμε τα **Records** του GDT και του ενός LDT που είναι ενεργός κάθε φορά έχουμε **16.384 Records**. Αυτό σημαίνει **16K Segments** σε ένα σύστημα. Πολλαπλασιάζοντας τα Segment με το μέγιστο μέγεθος του κάθε ενός από αυτά η συνολική **virtual memory** που μπορεί να χειριστεί ένας **intel Pentium 4** είναι **64Terabyte (16K x 4GB)**.

ΠΑΡΑΤΗΡΗΣΗ
Αν και η **virtual memory** που μπορεί να χειριστεί ένας επεξεργαστής σε Protected mode είναι **64TB** η **Linear Address** που μπορεί να παραχθεί είναι **μεγέθους μόνο 32bit**. Αυτό έχει σαν αποτέλεσμα οι φυσικές διευθύνσεις που παράγονται να **περιορίζονται μέχρι** τα **4GB**. Φυσικά τα **4GB** είναι υπέρ αρκετά ακόμα και σήμερα(2003) που οι απαιτήσεις σε μνήμη είναι πολλαπλάσιες από αυτές που υπήρχαν όταν πρωτοεμφανίστηκες ο 80386 εκτός από ειδικές περιπτώσεις όπως SQL εφαρμογές.

2.4 Ανακάτεμα του 16bit και του 32bit Protected Mode

Είναι σαφές ότι η λειτουργία του Virtual-8086 mode δεν θα μπορούσε να υλοποιηθεί από τον επεξεργαστή αν δεν μπορούσε να παρέχει **16bit διευθύνσεις** στο Protected mode. Κατά συνέπεια λοιπών το Protected mode μπορεί να είναι και 16bit. Με αυτόν το τρόπο μπορεί να τρέξουν εφαρμογές που είναι σχεδιασμένες να τρέχουν σε 16bit συστήματα.

Για να μπορεί όμως να εκτελεστή ο 16bit κώδικας σε ένα σύστημα που είδη βρίσκεται σε Protected mode επιλέχθηκε η λογική τα Segments να χρησιμοποιούν **κατάλληλους εκδικητές** όπως φαίνεται στο *σχήμα 2.4* που θέτουν το συγκεκριμένο Segment σε 16bit protected mode. Αυτό έχει σαν αποτέλεσμα το περιβάλλων που θα τρέξει μία εφαρμογή να μοιάζει με αυτό του Real–mode. Έτσι δημιουργούνται οι συνθήκες ώστε να μην αναγκαστεί ο επεξεργαστής να αλλάζει mode συνεχώς κάτι που σε ορισμένες περιπτώσεις θα ήταν αδύνατο.

Με την δυνατότητα αυτή του επεξεργαστή όταν ένα Segment είναι 16bit οι εντολές που περιέχει εκτελούνται σαν 16bit αλλιώς οι ίδιες εντολές εκτελούνται σαν 32bit. Για παράδειγμα η εντολή **"xor ax,ax"** και η **"xor eax,eax"** είναι μία 16 bit εντολή που το **opcode** που παράγει ένας Compiler και στις δύο περιπτώσεις είναι **"0x33 0xC0"**. Σε τέτοιες περιπτώσεις όπως το παράδειγμα είναι προφανές ότι ο μόνος τρόπος να μπορέσει ο επεξεργαστής να ξεχωρίσει πότε θα εκτελέσει μία εντολή σαν 16bit ή 32bit , όταν αυτός βρίσκεται σε Protected mode, είναι να ελέγξει προηγουμένως αν το Segment είναι σε 16bit ή 32bit mode αντίστοιχα.

Η δυσκολία του διαχωρισμού των εντολών από τον επεξεργαστή σε 16bit και 32bit καθιστά πολύ επικίνδυνη την ανάμιξη 16 και 32 bit κώδικά. Η επικινδυνότητα αυτή αναγκάζει τους προγραμματιστές συστημάτων αλλά και τους υπόλοιπους να **αποφεύγουν λύσης ανάμιξης** ειδικά όταν ο λόγος είναι απλά για να επαναχρησιμοποιηθούν modules παλαιότερων προγραμμάτων. Η μοναδική περίπτωση που δεν μπορεί να αποφευχθεί η ανάμιξη **16 και 32 bit κώδικα** είναι όταν ένα λειτουργικό σύστημα όπως τα **Windows** ή το **Linux** απαιτείται κατά την εκκίνηση του να θέσει τον επεξεργαστή από Real mode σε protected mode.

HACKERS NOTES

- Αυτό που πρέπει να συνειδητοποιήσει κανείς είναι ότι το Code Segments το Data Segment και το Stack Segment μία εφαρμογής μπορεί να έχει μέγεθος μέχρι **4GB**.
- Ο Χώρος του κάθε Segments μπορεί να προσπελαστεί **σαν ένας ενιαίος χώρος μνήμης** που αποτελείται από 32bit διευθύνσεις. Αν και οι διευθύνσεις αυτές στην πραγματικότητα είναι Offsets από την αρχή ενός Segment και όχι πραγματικές διευθύνσεις ο προγραμματιστής μπορεί για λόγους απλότητας **να τις θεωρεί σαν πραγματικές διευθύνσεις χωρίς καμία παρενέργεια.**

2.5 Η προστασία που προσφέρει το Protected Mode

Όπως είπαμε ο λόγος που χρησιμοποιείται το Protected mode δεν είναι μόνο για να μπορεί ο επεξεργαστής να χειρίζεται μεγάλη ποσότητα μνήμης αλλά και για να προσφέρει την κατάλληλη υποδομή να σχεδιαστούν και να υλοποιηθούν αξιόπιστα και ασφαλή λειτουργικά συστήματα. Η βασική υποδομή που παρέχει ένα intel επεξεργαστής είναι:

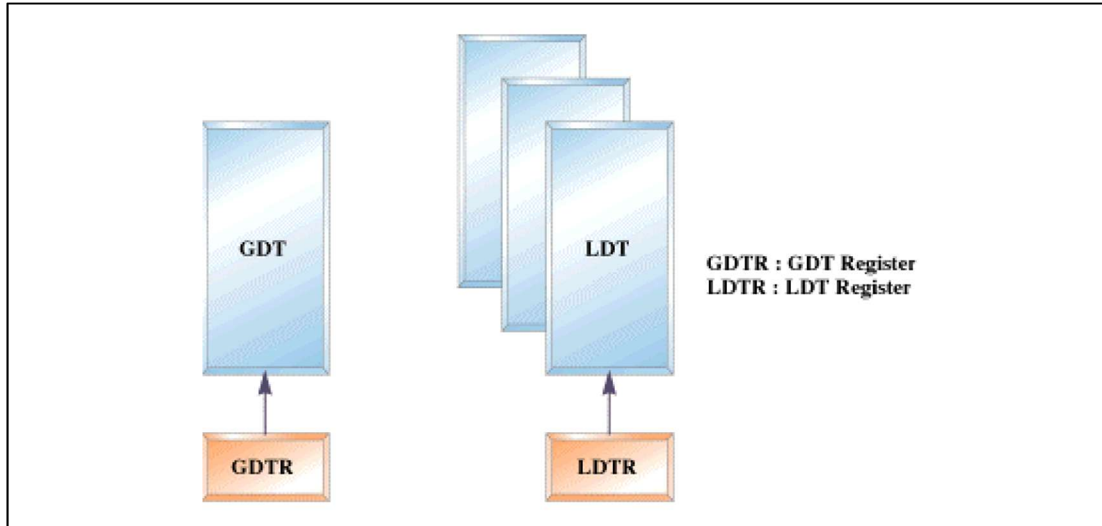
- Να εμποδίζεται ένα **task** να πειράζει εντολές και ένδικες που μόνο εξουσιοδοτημένες εφαρμογές επιτρέπεται να το κάνουν δηλαδή το λειτουργικό σύστημα.
- Να εμποδίζεται ένα **task** να αλλάξει τον κώδικα αλλά και τα δεδομένα άλλου task όταν και οι δύο βρίσκονται ταυτόχρονα στην μνήμη.
- Να εμποδίζεται ένα **task** να καλεί τον κώδικα του πυρήνα χωρίς την εξουσιοδότηση από τον ίδιο ή να μπορεί να επέμβει και να αλλάξει τον κώδικα και τα δεδομένα που ανήκουν στο ίδιο τον πυρήνα.

Για να ικανοποιηθούν οι τρεις παραπάνω δυνατότητες της υποδομής ενός intel επεξεργαστή γίνεται έλεγχος σε διάφορα επίπεδα από τον ίδιο τον επεξεργαστή. Η αλληλεπίδρασή του λειτουργικού συστήματος με αυτή την υποδομή είναι ιδιαίτερα έντονη και προς τις δύο κατευθύνσεις. Προς το παρόν εξετάζεται η πλευρά του επεξεργαστή ενώ στην επόμενη παράγραφο θα εξεταστεί η πλευρά του λειτουργικού συστήματος στο ίδιο θέμα (βλέπε Paging).

Το πρώτο επίπεδο προστασία που προσφέρεται είναι τα όρια του Segment. Όπως ειπώθηκε παραπάνω το **Offset** ενός προγράμματος **δεν μπορεί ποτέ** να ξεπεράσει το όριο που τίθεται από το **Limit** του Descriptor του συγκεκριμένου Segment. Όμως υπάρχει ακόμα ένα επίπεδο προστασίας πριν από αυτό του **Limit**. Συγκεκριμένα μία εφαρμογή δεν μπορεί να έχει πρόσβαση σε **LDT** που δεν τις ανήκει. Η συνέπεια αυτού είναι ότι **όχι μόνο απαγορεύεται η πρόσβαση** μίας εφαρμογής σε ένα Segment που δεν δεσμευτικέ για αυτή **αλλά ούτε την δυνατότητα να δει αυτό το Segment**. Το ποίο θα είναι αυτό το LDT υποδεικνύεται από τον **LDTR** καταχωριτή. Αυτός ο καταχωριστής θα δείχνει κάθε φορά τον LDT που είναι ενεργός. Φυσικά επειδή υπάρχει και ο **GDT** ο επεξεργαστής έχει ένα ειδικό καταχωριστή που δειχθεί σε αυτόν τον πίνακα και ονομάζεται **GDTR**.

Οι δύο αυτοί καταχωριστές είναι **48bit** και τα 32 bit χρησιμοποιούνται για να οδηγήσουν τον επεξεργαστή στην αρχή του GDT ή LDT αντίστοιχα ενώ τα υπόλοιπα 16 χρησιμοποιούνται σαν **όριο** του μεγέθους του πίνακα. Με αυτό το **όριο** δεν μπορεί κάποιο πρόγραμμα να διαβάσει εκτός των ορίων του εκάστοτε πίνακα. Παράλληλα συνεχίζει να επιτρέπεται σε ένα πρόγραμμα να αλλάξει την τιμή ενός Selector όπως και στην 16bit τεχνολογία.

Από τα παραπάνω είναι προφανές ότι και οι Selectors τελικά είναι **Offsets** ενώ η οι **GDTR** και οι **LDTR** υποδεικνύουν την **διεύθυνση βάσης** που ξεκινά ο αντίστοιχος πίνακας. Στο παρακάτω σχήμα *2.1* φαίνεται η χρησιμότητα των καταχωριτών GDTR και LDTR.



Σχήμα 7

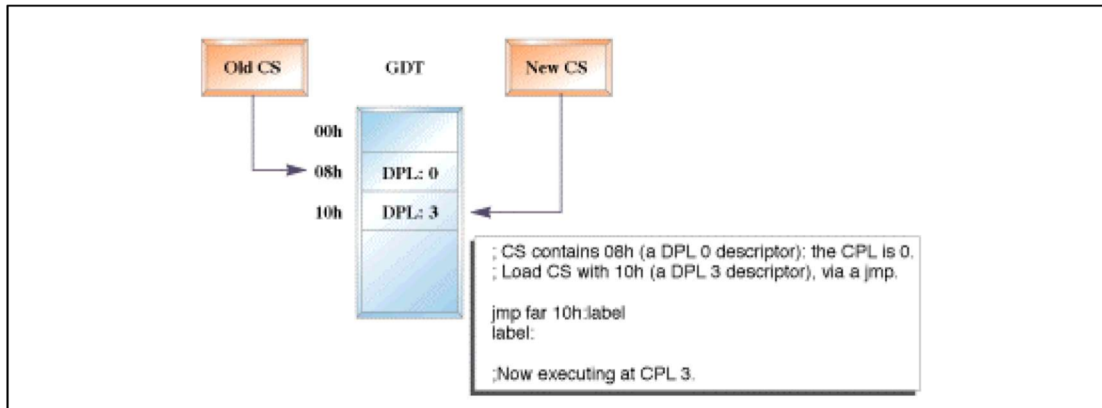
Το επόμενο επίπεδο προστασία που προσφέρει ο επεξεργαστής είναι μέσα σε κάθε Segment ελέγχοντας με διάφορους μηχανισμούς όπως το **TSS(Task State Segment)** τους **ενδείκτες ενός Descriptor**. Με τον έλεγχο αυτό δεν επιτρέπουν το να γραφτεί ένα Segment που είναι Read-only ή να κληθεί αυθαίρετα ένα Interrupt. **Ο σημαντικότερος όμως από όλους του ενδείκτες** είναι αυτός που χρησιμοποιείται από τον επεξεργαστή για να εφαρμόσει στα Segment τα **Επίπεδα Προνομίων(Privilege Level)**. Ο μηχανισμός των **επιπέδων προνομίων(Privileged Levels)** περιγράφεται στην παρακάτω παράγραφο.

2.6 Τα Privileged Levels η βασική υποδομή για την υλοποίηση των σύγχρονων λειτουργικών συστημάτων

Όπως ειπώθηκε ένα **tasks** δεν μπορεί να έχει πρόσβαση σε Segments που δεν του ανήκουν την στιγμή που δεν μπορεί να δει LDT διαφορετικό από το δικό του. Εκτός όμως από αυτό υπάρχει και το GDT που είναι **καθολικά(Global)** προσβάσιμο από όλα τα Task. Για να υπάρχει έλεγχος πρόσβασης και σε αυτή την περίπτωση χρησιμοποιείται ένας άλλος μηχανισμός προστασίας που λέγεται **Privilege Level**.

Τα **Privilege Levels** που μπορεί να έχει ένα task είναι 4. Τα **Privilege Levels** ξεκινούν από το **0 (most privileged)** και φτάνουν μέχρι το **3 (least privileged)**. Όταν ένα task έχει **PL 0** τότε μπορεί να κάνει τα πάντα από το να αλλάξει τα interrupt του συστήματος μέχρι και να αλλάξει τους Descriptors. Κανένα από τα **1,2 ή 3** επίπεδο δεν έχει τόσα πολλά δικαιώματα. Τα επίπεδα αυτά έχουν άμεση σχέση με την κατάσταση που βρίσκεται ο επεξεργαστής αλλά και με την το κάθε Segment. Έτσι για παράδειγμα όταν ο επεξεργαστής μεταβεί από real-mode σε protected mode η πρώτη εφαρμογή που θα τρέξει θα έχει **PL 0**. Αυτό είναι πολύ λογικό γιατί το πρώτο πράγμα που θα τρέξει είναι το **λειτουργικό σύστημα** που προφανώς θα πρέπει να έχει το **υψηλότερο επίπεδο προνομίων**. Στην συνέχεια το λειτουργικό σύστημα θα αποφασίσει σε τι επίπεδο δικαιωμάτων θα τρέχουν *οι εφαρμογές(συνήθως PL 3)* των χριστών ή άλλα προγράμματα όπως οι *Drivers(συνήθως PL 1 ή 2)*.

Όσο αφορά την σχέση του PL με το Segment έχει πολύ ενδιαφέρον. Ένα Task δεν μπορεί να έχει πρόσβαση σε ένα Segment που έχει προνόμια μεγαλύτερα από την ίδια την εφαρμογή. Για να μπορέσει να κάνει αυτό τον έλεγχο επεξεργαστής χρησιμοποιεί τα **DPL bits** του **Descriptor**. Αν το Segment έχει **DPL 0** κανένα άλλο Task εκτός από αυτό που έχει **προνόμιο επιπέδου 0** δεν μπορεί να αλληλεπιδράσει με αυτό το Segment. Αντίθετα ένα Segment με DPL 3 είναι προσβάσιμο από κάθε Task ασχέτως δικαιωμάτων. Τα δικαιώματα που έχει ένα Task τα κρατά το **CPL(Current Privilege Level) που συσχετίζεται άμεσα με τον εκτελέσιμο κώδικα του Task**. Φυσικά όλα αυτά ισχύουν για όλα τα LDT και GDT αλλά εκεί που έχουν ιδιαίτερη σημασία είναι στον GDT εφόσον **τα Segments που περιγράφονται από τους Descriptor ενός GDT εν γένη είναι προσβάσιμα από όλα τα Task**. Στο παρακάτω σχήμα φαίνεται η χρήση του DPL.



Σχήμα 8

Για αν μπορέσει να καταλάβει ο επεξεργαστής αν το PL που θα μεταβεί ένα Task είναι επιτρεπτό συγκρίνει το **CPL** με το **RPL** που βρίσκεται στον **CS selector του task**. Το ερώτημα είναι τι θα γίνει το task ζητήσει πρόσβαση με ένα CS που έχει διαφορετικά δικαιώματα από το προηγούμενο. Στην πράξη ο επεξεργαστής επιτρέπει την **μετάβαση των δικαιωμάτων προς μια μόνο κατεύθυνση δηλαδή από τα υψηλότερα προς τα χαμηλότερα δικαιώματα.** Αυτό όπως φαίνεται στο παράδειγμα σημαίνει ότι **όταν ένα Task με δικαιώματα 0** αρχίζει να **διαβάζει κώδικα απο Segment με χαμηλότερο DPL (1,2,3) δεν μπορεί να επιστρέψει και να διαβάσει κώδικα υψηλότερο DPL**. Άρα όταν το λειτουργικό σύστημα ενεργοποιήσει το User Space που τα Segment του έχουν DPL 3 καμία εφαρμογή που θα τρέξει στο User Space δεν μπορεί να αποκτήσει δικαιώματα PL 0 ή αλλιώς *καμία εφαρμογή που εκτελεστικές σε Segment με DPL 3 δεν μπορεί να έχει πρόσβαση σε Segment με DPL 0 ακόμα και αν αρχικά είχε αυτό το **δικαίωμα(Privilege)**.*

2.7 Το Paging των λειτουργικών συστημάτων

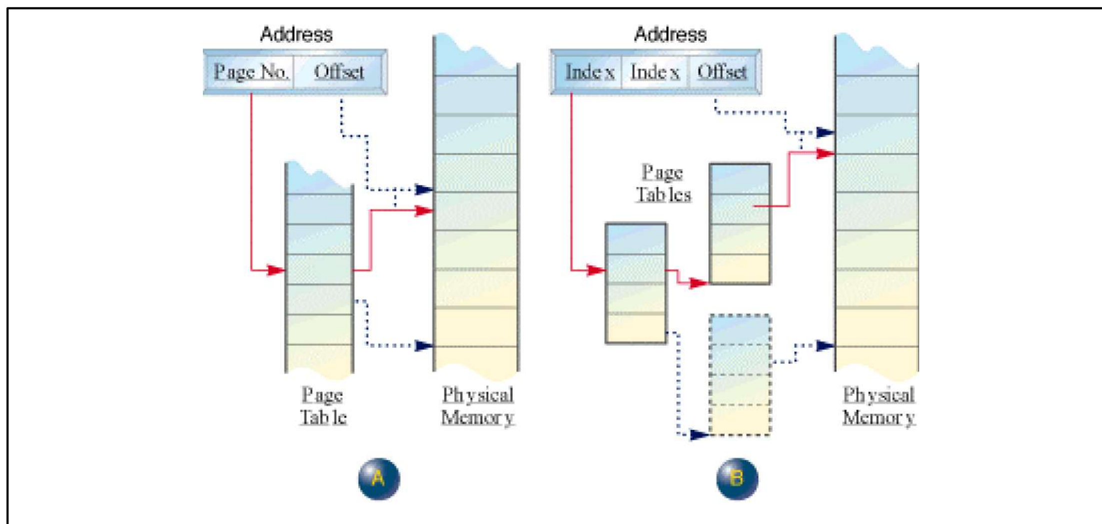
Όπως είπαμε ο σκοπός του Protected mode ήταν να προσφέρει την κατάλληλη υποδομή ώστε να μπορέσουν να σχεδιαστούν λειτουργικά συστήματα που θα καλύπτουν υψηλές απαιτήσεις. Με το Protected mode λοιπών μπόρεσε με πολύ αποδοτικό τρόπο να σχεδιαστεί μία υποδομή που αφενός προσφέρει διαχείριση πολύ μεγάλης ποσότητας μνήμης(4GB) αφετέρου την υποδομή προστασίας ώστε να μην μπορεί η εφαρμογή ενός χρήστη εκούσια ή μη να καταστρέψει όλο το σύστημα. Το πρόβλημα που προκύπτει από αυτή την υποδομή είναι ότι αφενός δεν θέλουμε να περιορίζουμε το Segment μίας εφαρμογής στο 1MB αφετέρου στις περισσότερες περιπτώσεις δεν χρειάζεται Segment που να φτάνει τα 4GB.

Η λύση του προβλήματος με επικαλυπτόμενα Segments είναι εφικτή αλλά έχει ένα μεγάλο μειονέκτημα. Το μειονέκτημα είναι ότι **ο επεξεργαστής θα πρέπει να αλλάζει συνεχώς τις τιμές των Segment Registers όπως έκανε και στην 16bit τεχνολογία** κάτι που δεν είναι καθόλου αποδοτικό. Η λύση στο πρόβλημα αυτό είναι το Paging.

Το Paging είναι ένας διαφανής μηχανισμός που επιτρέπει σε μία εφαρμογή να νομίζει ότι της διαθέτονται 4GB μνήμης ενώ παράλληλα επιτρέπει στο λειτουργικό σύστημα να έχει μόνο ένα μικρό κομμάτι στην μνήμη από τον συνολικό κώδικα που της εφαρμογής αυτής. Το κομμάτι αυτό του κώδικα που βρίσκεται στην μνήμη είναι κάθε φορά αυτό που χρειάζεται να εκτελεστεί ενώ τα υπόλοιπα, αν δεν υπάρχει χώρος στην μνήμης(αλλά συνήθως και όταν υπάρχει), βρίσκονται στο **swap partition** ή σε κάποιο **page file** του δίσκου.

Το βασικό πλεονέκτημα του Paging είναι ότι κομματιάζει το Segment που δεσμεύεται για την εφαρμογή σε μικρότερα κομμάτια από 4GB, συνήθως **512byte ως 8KB (στους intel είναι 4K)**, και μπορεί να τα τοποθετήσει στην μνήμης σε τελείως διαφορετικές περιοχές από αυτές που προσδιορίζονται από τον επεξεργαστή(βλέπε Descriptors). Ο κατακερματισμός που προκαλεί το Paging είναι πολύ σημαντικός γιατί **αυξάνει την χρηστικότητα** της μνήμης σε πολύ μεγάλο βαθμό χωρίς να αναγκάζει το σύστημα να παράγει συνεχώς νέα Segments. Τέλος πριν συνεχίσουμε να δούμε με μεγαλύτερη λεπτομέρεια το Paging θα τονιστεί μια ακόμα φορά ότι **όλη η διαδικασία γίνεται τελείως διάφανα για το πρόγραμμα**.

Στην πράξη λυπών το μοντέλο του Paging δεν διαφέρει πολύ από αυτό του Segmentation που προσφέρει ο επεξεργαστής μόνο που αυτή την φορά όλους τους ελέγχους τους κάνει το λειτουργικό σύστημα. Επίσης αντί **η μνήμη να χωρίζεται** σε Segments **χωρίζεται σε Pages (σελίδες)**. Σημαντικό εδώ είναι να σημειωθεί ότι για το Paging ο επεξεργαστής έχει καταχωριτές που αποτελούν την υποδομή για αυτό. **Αν δεν υπήρχε η Hardware υποδομή δεν θα μπορούσε να υλοποιηθεί το Paging**. Το μοντέλο που συνήθως χρησιμοποιούνται στα λειτουργικά συστήματα είναι αυτά που παρουσιάζονται στο παρακάτω σχήμα.



Σχήμα 9

Όπως και στο Segmentation έτσι και εδώ υπάρχουν κατάλληλοι πίνακες που περιέχουν τις **διευθύνσεις βάσεις** κάθε **σελίδας(Page)** και ένα σύνολο από **Flags(ένδικες)** που αφορούν τα δικαιώματα χρήσης.

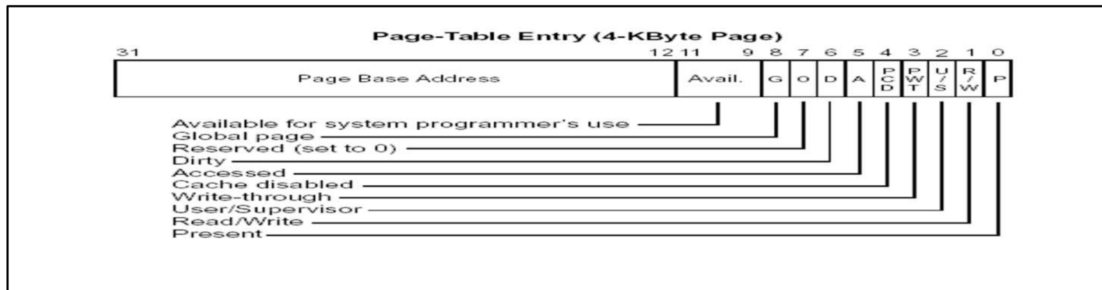
Στο Paging είναι ότι η **Linear Address** που θα παραχθεί αντί να χρησιμοποιηθεί σαν **φυσική διεύθυνση** να χρησιμοποιηθεί σαν **Index** σε ένα άλλο πίνακα που θα οδηγήσει τελικά στην **φυσική διεύθυνση** που διαθέτει εκείνη την στιγμή το λειτουργικό σύστημα για την εφαρμογή.

Στην πράξει για να συνέβαινε αυτό που περιγράφεται παραπάνω θα έπρεπε να δημιουργηθεί ένας πίνακα μεγέθους **4GB** για να επαναπροσδιορίζει την **φυσική διεύθυνση** που διαθέτεί κάθε στιγμή το λειτουργικό σύστημα για την Linear Address που παράγει ο επεξεργαστής. Για να αποφευχθεί αυτό η **Linear Address** χωρίζεται σε διάφορα κομμάτια από 2 ως 4 στο Linux ώστε να παράγονται μικρότεροι πίνακες. Μάλιστα οι πίνακες αυτοί όπως φαίνεται και στο παραπάνω σχήμα μπορεί να βρίσκονται και η ίδιοι σε Pages που αποθηκεύονται στον δίσκο (π.χ. στο Page file) όταν αυτοί δεν είναι απαραίτητοι.

Στο παράδειγμα A του σχήματος 2.9 φαίνεται ότι η Linear Address μπορεί να χωριστεί σε δύο τμήματα. Το πρώτο προσδιορίζει την εγράφη στον πίνακα των σελίδων και το δεύτερο προσδιορίζει το **Offset** δηλαδή, όπως και στο segmentation, την απόσταση από την Base Address που περιέχεται στην εγράφη. Την εγράφη του κάθε πίνακα την λεμε **PTE(Page Table Entry)** ενώ τον πίνακα τον λέμε **Page Table (PT)**.

Συνήθως για λόγους εξοικονόμησης μνήμης τα λειτουργικά συστήματα χωρίζουν την Linear Address σε 3 ή 4 μέρη. Έτσι όπως στο παράδειγμα B του σχήματος 2.9 η διεύθυνση χωρίζεται σε 3 μέρη το πρώτο μέρος οδηγεί σε ένα πίνακα σελίδων που περιέχουν πίνακες σελίδων για αυτό τον πίνακα αυτό τον λέμε **Page Directory Table** ενώ κάθε γραφή μέσα σε αυτόν την λεμε **PDE(Page Directory Entry)**. Το δεύτερο κομμάτι της linear address του παραδείγματος είναι στην πράξει το **Offset** που υποδεικνύει μέσα στον πίνακα **PT** την εγράφη PDE. Συνεπώς αυτό το δεύτερο κομμάτι της Linear Address χρησιμοποιείται σαν Index στον PT πίνακα. Με αυτό τον τρόπο προσδιορίζεται το PDE που θα επιστρέψει με την σειρά του την **Base address** της σελίδας που βρίσκεται μέσα στην Φυσικής μνήμης. Κατά τα γνωστά η

Base Address θα προστεθεί με το **Offset** ώστε να προσδιοριστεί η φυσική διεύθυνση μνήμης που τελικά θα έχει πρόσβαση το πρόγραμμα. Το Offset αυτό αντιστοιχεί στο τρίτο κομμάτι της Linear Address όπως φαίνεται στο σχήμα 2.9. Παρακάτω φαίνεται η εικόνα ενός **PTE.**

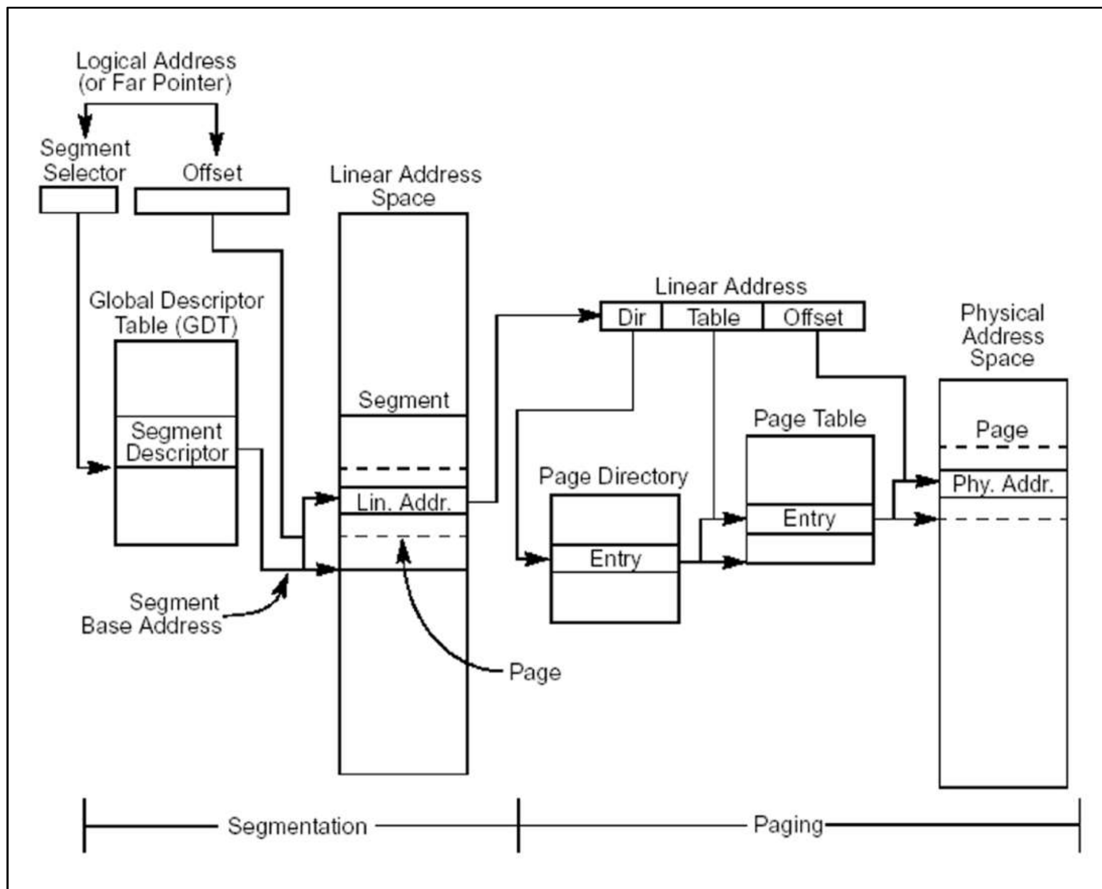


Σχήμα 10

Όπως και στο Segmentation έτσι και στο Paging χρειάζεται ένας τουλάχιστον καταχωριστής που θα δείχνει συνεχώς την διεύθυνση μνήμης που βρίσκεται κάθε **PT** για αυτό το σκοπό ένα intel επεξεργαστής διαθέτει τον καταχωριτή CR3, του σχήματος 2.2, για αυτό το σκοπό. Στην περίπτωση που χρησιμοποιείται και ένα **PDE** τότε ένα καταχωριτής ο **CR3** πρέπει να δείχνει στην αρχή αυτού του πίνακα.
Από όλα τα παραπάνω συμπεραίνουμε ότι τι Paging είναι ο μηχανισμός που δεν αντικαθιστά το Segmentation αλλά το επεκτείνει. Με αυτό τον τρόπο το Paging θα χρησιμοποιείται για την διαχείριση τον συνηθισμένων αναγκών ενός 32bit συστήματος ενώ όταν η ανάγκες σε μνήμη **αφήνοντας ανενεργό το Segmentation** ενώ το τελευταία θα χρησιμοποιείται όταν οι ανάγκες ανά Task ξεπεράσουν τα 4 GB. Στο *Σχήμα 2.11* φαίνεται η σχέση του Segmentation με το Paging.

HACKERS NOTE

Από όλα τα παραπάνω το σημαντικότερο που πρέπει να θυμάται κανείς είναι ότι λόγο της χρήσης του Paging είναι για να μην χρησιμοποιούνται πολλά Segments. Αυτό σημαίνει στην πράξη ότι ένα πρόγραμμα μπορεί να μοιράζεται το ίδιο Segment με άλλα προγράμματα καταργώντας πολλές από την δυνατότητες ασφάλειας που υπάρχουν όταν το Segmentation χρησιμοποιείται ενεργά. Φυσικά αυτό έχει το πλεονέκτημα ότι απλουστεύει ακόμα περισσότερο τον προγραμματισμό, ειδικά όταν χρειάζεται να υπάρχουν διαμοιραζώμενες βιβλιοθήκες όπως τα DLL, αλλά παράλληλα αφήνει τα πράγματα στις δυνατότητες ασφάλειας του λειτουργικού συστήματος. Το πρόβλημα είχε προβλεφθεί από του κατασκευαστές συστημάτων και για αυτό το λόγο τα περισσότερα από τα **Flags ασφάλειας** των Segment έχουν μεταφερθεί στα Flags Ασφάλειας των PDE και PTE όπως φαίνεται στο σχήμα 2.10. **Παρά την ασφάλεια που παρέχεται από τα Pages μια εφαρμογή δεν περιορίζεται να διαβάζει μόνο τις δικές της σελίδες όπως συμβαίνει με τα Segments αλλά μπορεί να διαβάσει ή και να αλλάξει τα περιεχόμενα σελίδων που δεν ανήκουν σε αυτή**.



Σχήμα 11

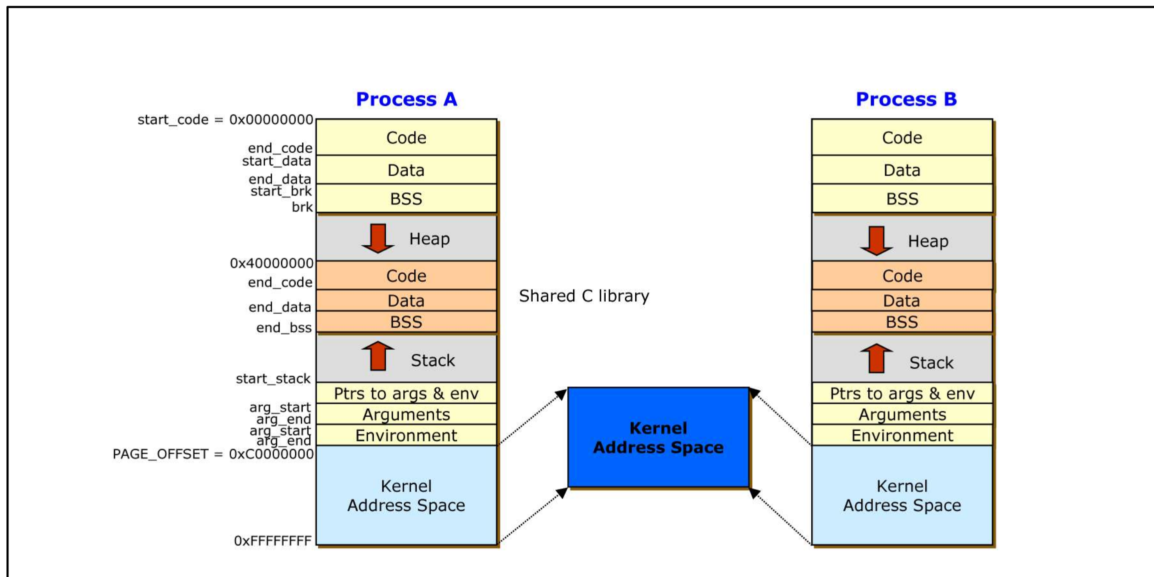
Παρακάτω θα παρουσιαστεί πιο είναι το μοντέλο διαχείρισης της μνήμης που έχουν τα λειτουργικά συστήματα Linux και Window.

3 Η Αρχιτεκτονική του Linux και των Windows MS-Windows (NT generation).

Η Αρχιτεκτονική των δύο αυτών λειτουργικών συστημάτων έχει τεράστιες διαφορές αλλά και πολλή σημαντικές ομοιότητες. Πολλές φορές μάλιστα η διαφορές τους είναι μόνο σε επίπεδο υλοποίησης. Αυτά τα κοινά χαρακτηριστικά θα εξεταστούν εδώ με σκοπό να γίνει κατανοητά στην πράξη τα χαρακτηριστικά που διαθέτει σήμερα ένα 32bit **Λειτουργικό Σύστημα**. Οι επεξηγήσεις θα γίνονται με βάση την αρχιτεκτονική Linux αλλά η δυνατότητες αυτές υπάρχουν και στα δύο συστήματα. Τέλος όπου υπάρχει διαφορά θα επισημαίνεται.

3.1 To Flat Memory Addressing

Όπως είπαμε κατά κανόνα τα λειτουργικά συστήματα σήμερα χρησιμοποιούν το Paging και μόνο σε εξαιρετικές περιπτώσεις το Segmentation. Έτσι το Linux και τα Windows έχουν επιλέξει ένα μοντέλο για την διαχείριση της μνήμης που **απλουστεύει την υλοποίηση** πολλών **από τα χαρακτηριστικά που παρέχουν** όπως τα DLL και τα SO. Το μοντέλο αυτό λέγεται **Flat Memory Addressing** και φαίνεται στο παρακάτω σχήμα.

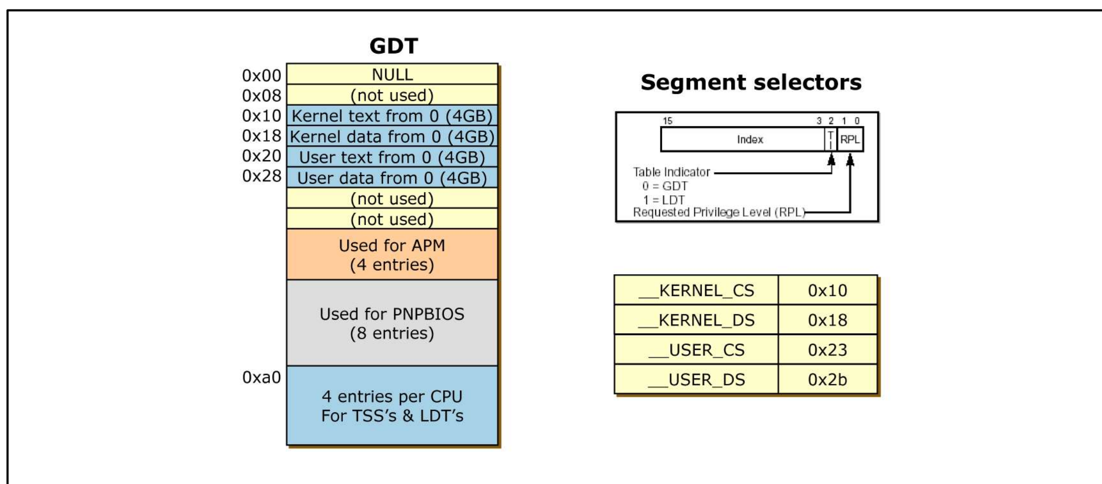


Σχήμα 12

Στο **Flat Memory Addressing συμβαίνουν** τα έξείς:

- Ο χώρος την μνήμης χωρίζεται σε δύο **Περιοχές** που η μια ονομάζεται **Kernel Space** και χρησιμοποιείται από τον πυρήνα του συστήματος δεύτερη **User Space και χρησιμοποιείται από τις εφαρμογές των χρηστών**. Από τα Segment αυτά το Kernel Space έχει DPL 0 ενώ το User Space έχει DPL 3.
- Κάθε **Processes** μιας εφαρμογής, όπως φαίνεται στο σχήμα 2.12, στο User Space μοιράζεται ακριβώς τις ίδιες διευθύνσεις μνήμης με όλα τα υπόλοιπα Processes της ίδιας ή και άλλων εφαρμογών. Ο συνολικός χώρος που μπορεί να καταλάβει μια εφαρμογή είναι 4GB όσο τα όρια του Segment. Ο διαχωρισμός για το ποίες φυσικές διευθύνσεις καταλαμβάνει κάθε Process Βασίζεται στο Paging.

Είναι σημαντικό να που με ότι οι **Περιοχές Kernel και User Space** αποτελούνται από δύο Segments έκαστη. Το ένα Segment είναι Read-only και χρησιμοποιείται για τον κώδικα ενώ το δεύτερο χρησιμοποιείται για τα **δεδομένα(Data) την στοίβα(Stack) και το σωρό(Heap)**. Η εικόνα του GDT στο Linux όταν υλοποιεί το Flat Memory Addressing είναι αυτή που φαίνεται στο *σχήμα 2.13*. Στην περίπτωση των **MS-Windows** τα Segments που χρησιμοποιούνται για τις εφαρμογές μπορεί να είναι **μόνο μέχρι 2GB**.



Σχήμα.13

Από όλα όσα έχουν ειπωθεί μέχρι εδώ συμπεραίνουμε ότι ο προγραμματισμός των εφαρμογών 32bit με το **Flat Memory Addressing** γίνεται εξαιρετικά απλός και φυσικά αποτελεσματικός. Με το **Flat Memory Addressing** έχουμε τα παρακάτω πλεονεκτήματα:

- Οι CS DS και SS παραμένουν σταθεροί σε όλη την διάρκεια που ένα **Process** τρέχει σε User Space. **Αυτό σημαίνει ότι όλες οι εφαρμογές χρησιμοποιούν το ίδιο Base Address όταν αυτές τρέχουν σε User Space**. Αυτό επίσης σημαίνει ότι το **Offset** που χρησιμοποιείται μέσα σε ένα πρόγραμμα *μπορούμε να θεωρήσουμε ότι αναφέρεται σε φυσικές διευθύνσεις* αν και στην πραγματικότητα δε συμβαίνει αυτό.
- Αυτό το μοντέλο διαφάνειας για τον προγραμματιστή υποστηρίζεται ακόμα περισσότερο από το Paging του **Flat Memory Addressing** που οδηγεί **διαφανώς** ένα πρόγραμμα σε διαφορετικό **Offset** από αυτό που πραγματικά ζήτησε το πρόγραμμα του χωρίς όμως να επηρεάζει την λειτουργία του προγράμματος.
- **Ο προγραμματιστής μπορεί με ασφάλεια να ξεχάσει όλη την προηγούμενη πολυπλοκότητα** όταν προγραμματίζει σε 32 bit περιβάλλων **και να θεωρεί ότι έχει ένα Ενιαίο(Flat) χώρο των 4GB (2GB στα Windows)** για την εφαρμογή του.
- Οι διεργασίες με το **Flat Memory Addressing** μπορούν να αλληλεπιδρούν μεταξύ τους χωρίς να χρειάζονται **Calling Gates** ή άλλοι πολύπλοκοι μηχανισμοί για να υποστηρίξουν αυτή την δυνατότητα.

Παρακάτω θα παρουσιαστεί το ELF μοντέλο του Linux που μοιάζει αρκετά με το PE των Windows. Οι τεχνολογίες αυτές βασίζονται στο **Dynamic Linking** που θα ήταν εξαιρετικά πολύπλοκο να υλοποιηθεί αν τα λειτουργικά συστήματα χρησιμοποιούσαν διαφορετικό μοντέλο από το **Flat Memory Addressing**.

HACKERS NOTE
Κατά το **Flat Memory Addressing** συμβαίνουν δύο σημαντικά πράγματα που μπορούν να χρησιμοποιηθούν και από πλεονεκτήματα να γίνουν μειονεκτήματα:

- Οι εφαρμογές μπορούν να αλλάξουν τα δεδομένα άλλων εφαρμογών και να διαβάσουν των κώδικα από σελίδες που δεν πρέπει.
- Οι στοίβες που δεσμεύονται για τις εφαρμογές **πάντα θα βρίσκονται στην ίδια περιοχή μνήμης όπως φαίνεται στο σχήμα 2.12**. Το σημαντικότερο όμως μειονέκτημα είναι ότι τα δεδομένα την στοίβας ενώς προγράμματος μπορεί να τα δει ή να τα αλλάξει ένα άλλο πρόγραμμα. Αυτό έχει τις εξής συνέπειες:
  - Να διαβαστούν πληροφορίες που δεν πρέπει όπως για παράδειγμα κάποιο Password.
  - Να υπολογιστούν τα **Offset** που θα χρησιμοποιήσει ένα άλλο πρόγραμμα.
  - Να αλλαχτούν πληροφορίες που αφορούν την ροή ενός προγράμματος από ένα άλλο πρόγραμμα.
- Η περιοχή των **δεδομένων(Data region)** βρίσκεται στην ίδια μοίρα με αυτή της στοίβας.

3.2 Τα εκτελέσιμα αρχεία των Windows και του Linux.

3.2.1 Γενικά

Για να εκτελεστεί ένα πρόγραμμα το σύστημα δεσμεύει μνήμη σε τρεις βασικές περιοχές.

1. **Text ή code region**.
2. **Data region ή heap**.
3. **Stack region**.

Text ή Code region

Στην **text ή code region** αποθηκεύονται οι εντολές του προγράμματος. Αυτή η περιοχή είναι read-only δηλαδή δεν έχουμε δυνατότητα μόνο να γράψουμε σε αυτήν. Την πρώτη διεύθυνση μνήμης από την οποία αρχίζει την αυτή η περιοχή την περιέχει ο **CS register** δηλαδή ο Code Segment καταχωριστής **στο Real-mode ενώ στο Protected mode** δείχνει τον **Descriptor** που την περιέχει.

Data region

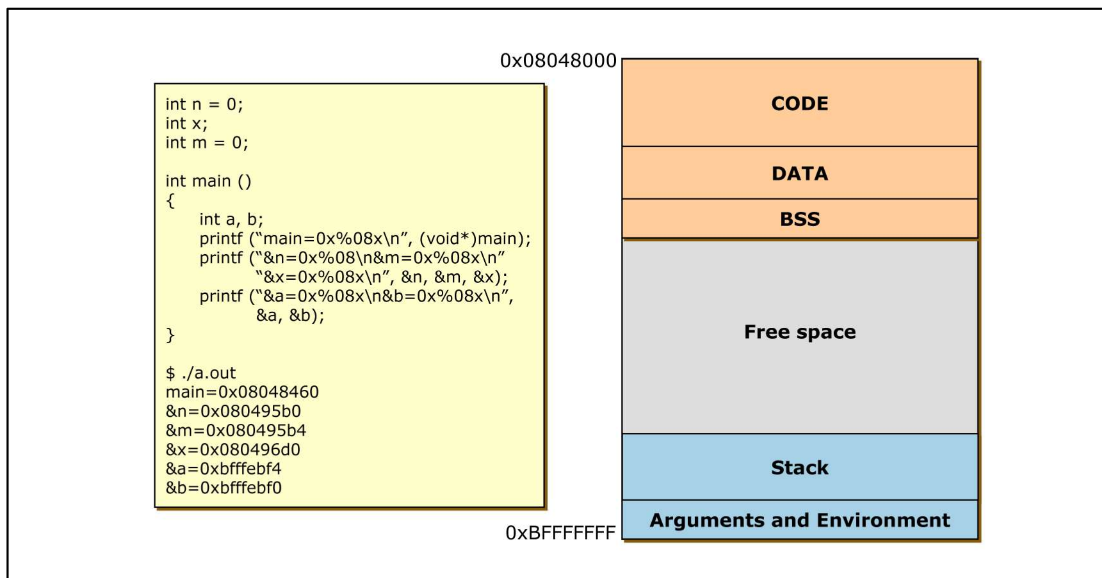
Η data region αποτελείται από **3 Sections** το **Data Section** το **BSS** και το **Heap Section**. Στο Data Section αποθηκεύονται στατικές μεταβλητές οι Pointer με σταθερό περιεχόμενος. Στο BSS κατά κανόνα υπάρχουν Pointer προς το Heap. Στο Heap δεσμεύονται οι static μεταβλητές του συστήματος και το μέγεθος της περιοχής αυτής είναι μεταβλητό. Το μέγεθος του αλλάζει από την system call brk(2). Η πρώτη διεύθυνση μνήμης που ξεκινά το Data region είναι αποθηκευμένη στο καταχωριστή **DS register** δηλαδή τον Data Segment καταχωριστή στην περίπτωση **στο Real-mode ενώ στο Protected mode** δείχνει τον **Descriptor** που την περιέχει. Αν ο χώρος του σορού(Heap) για μια διεργασία εξαντληθεί τότε η διεργασία σταματά να εκτελείται από τον χρονοδρομολογητή (time scheduler) και ξαναεκτελείται(Rescheduled) έχοντας περισσότερο διαθέσιμο σορό(Heap).

Stack region

Στην **Stack region** είναι ο χορός της μνήμης που αναφέραμε παραπάνω και θα μας απασχολήσει στην πτυχιακή αυτή. Εδώ είναι ο χορός που καταγράφονται dynamic allocating μεταβλητές που χρησιμοποιούνται εσωτερικά στις functions/procedures. Ο τρόπος λειτουργίας της στοίβας είναι ο γνωστός **LIFO(Last In First Out)** δηλαδή ότι δεδομένα μπαίνουν σε αυτή τελευταία αυτό θα βγουν πρώτα. Ακόμα εδώ αποθηκεύεται η **τιμή** που είχε ο **EIP(Instruction Pointer)** πριν από την κλίση μίας συνάρτησης. Η τιμή αυτή ονομάζεται και **RA(return address)** δηλαδή **διεύθυνση επιστροφής** επειδή από αυτή την διεύθυνση συνεχίσει ροή του προγράμματος **κατά την επιστροφή της συνάρτηση** μετά το πέρας της εκτέλεσης της συνάρτησης. Η πρώτη διεύθυνση μνήμης από οπού ξεκινάει η περιοχή της στοίβας αποθηκεύεται στον **SS** δηλαδή στον Stack Segment register. Σε πολλές υλοποιήσεις σε διάφορες πλατφόρμες **ο σωρός και η στοίβα είναι ο ίδιος χώρος** μέσα στην μνήμη.

3.1.2 Το ELF και το PE

Ακριβώς αυτή την δομή έχουν τα ELF και τα PE όταν αυτά Φορτώνονται στην μνήμη και αυτή η δομή φαίνεται στο παρακάτω σχήμα :



Σχήμα 2.14

Με το απλό πρόγραμμα C που υπάρχει μέσα στο σχήμα μπορούμε να διαπιστώσουμε ότι σε όσες φορές και να τρέξουμε το πρόγραμμα οι διευθύνσεις των μεταβλητών που θα πάρουμε για τους διάφορους τύπους μεταβλητών θα ανήκουν πάντα στις ίδιες περιοχές μνήμης. Έτσι για παράδειγμα μια **Local Variables** θα βρίσκεται πάντα σε περιοχή μνήμης **0xbfff*xxxx*** ανεξάρτητα από το πόσα προγράμματα τρέχουν παράλληλα με αυτό το πρόγραμμα κατά συνέπεια ανεξάρτητα από το **πόσες στοίβες** βρίσκονται μέσα στην μνήμη. Αυτό οφείλεται στο γεγονός της χρήσης του **Flat Memory Addressing** από το λειτουργικό σύστημα.

Επίσης πρέπει να παρατηρήσουμε ότι ο χώρος Free Space που φαίνεται παραπάνω θα χρησιμοποιηθεί για τρεις λόγους όταν το ELF φορτωθεί στην μνήμη. Αρχικά εκεί θα δεσμευτεί ο σορός(Heap) του Data Region ώστε να **μπορεί να δεσμεύεται δυναμικά χώρος** για αυτά. Από αυτό τον χώρο θα δεσμεύει τα byte που της χρειάζονται και η **στοίβα(Stack)**. Τέλος **ανάμεσα στην στοίβα και στον σωρό** θα βρίσκονται οι βιβλιοθήκες που χρησιμοποιεί ένα πρόγραμμα όταν χρησιμοποιείται το **Dynamic Linking**.

3.1.3 Το Dynamic Linking και η χρησιμότητα του GOT

Παλιότερα πριν την εμφάνιση της 32bit τεχνολογίας οι εφαρμογές χτίζονταν από τον Compiler που έπρεπε να συνδέσει ένα σύνολο μικρών προγραμμάτων γραμμένα σε C που αποτελούσαν την εφαρμογή. Με το Compilation αυτό που γινόταν ήταν να **παράγονται μερικά object Files** όπως συμβαίνει και σήμερα. Τα object Files για να μπορούν να επικοινωνούν μεταξύ τους είχαν εντολές που συσχέτιζαν τα object Files μεταξύ τους. Οι πληροφορίες αυτές έμπαινα σε άλλα Object Files που το ονομάζουμε και **σήμερα reloc (relocation) object**.

Για να ολοκληρωθεί η παραγωγή του εκτελέσιμου αρχείου μετά από τον Compiler χρησιμοποιούταν ο **Linker** όπως συμβαίνει και σήμερα. Ο Linker αυτό που έκανε **ήταν να συνδέσει** όλα τα Object Files μεταξύ τους. Για να το κάνει αυτό «έτρεχε μέσα» στο **reloc objects** και τα συμπλήρωνε όπου χρειαζόταν με τις εντολών που του έλειπαν. Οι εντολές που έλειπαν δεν ήταν τίποτα παραπάνω από τις εντολές του κάθε ενός από τα Objects που αποτελούσαν την εφαρμογή. Έτσι γινόταν η σύνδεση όλων το ν Object files με την βοήθεια του **Linker**. Αυτό ακριβώς συμβαίνει και σήμερα αν ζητήσουμε από το GCC ή οποιοδήποτε C Compiler να κάνει **static Linking**.

Σήμερα όμως αυτό που γίνεται είναι να χρησιμοποιείται το **Run-Time Linking** δηλαδή να γίνεται η σύνδεση των Object Files μεταξύ τους την ώρα που η ίδια η εφαρμογή τρέχει. Η διαδικασία λέγεται **Dynamic Linking**.

Το **Dynamic Linking** χρησιμοποιείται από τα ELF files του Linux και από τα PE files των windows. Πρακτικά σημαίνει ότι την τεχνολογία αυτή την χρησιμοποιούν τα .exe αρχεία των windows που δεν μπορούν να τρέξουν στο DOS, και τα Binary αρχεία του Linux που έχουν δικαίωμα **εκτέλεσης(+x)**. Ο λόγος που χρησιμοποιείται το **Dynamic Linking** είναι για να φορτώνονται οι λεγόμενες δυναμικές βιβλιοθήκες που στα windows λέγονται DLL ενώ στο Linux λέγονται SO.

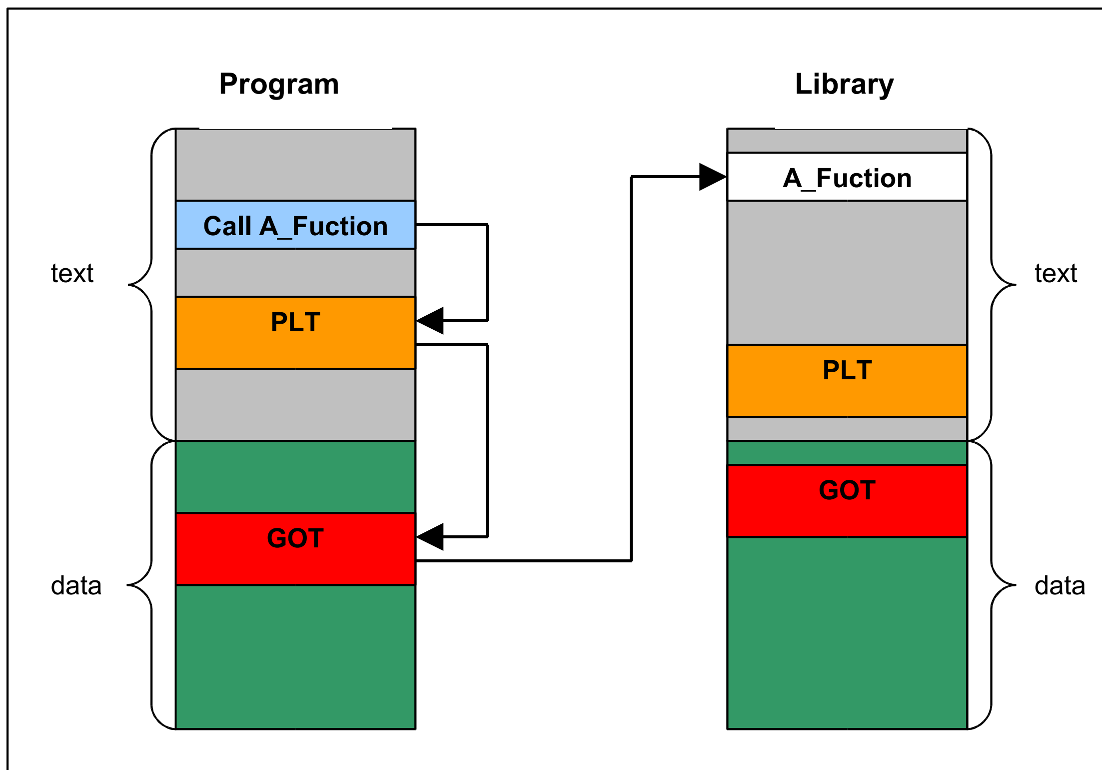
Στην περίπτωση λοιπών που μία βιβλιοθήκη είναι εξαρχής φορτωμένη στην μνήμη δεν χρειάζεται να ξαναφορτωθεί αλλά απλά να βρεθεί ένα τρόπος με το οποίο να συνδεθεί η βιβλιοθήκη αυτή να συνδεθεί στην εφαρμογή που την χρειάζεται. Για να το κάνει αυτό το κάθε ένα από τα λειτουργικά συστήματα χρησιμοποιεί την δικιά του τεχνολογία. Στην προκείμενη περί περίπτωση θα δούμε συνοπτικά τον μηχανισμό του Linux.

Το πρόβλημα στο Dynamic Linking είναι ότι δεν μπορεί ο Linker να γνωρίζει από πριν την σχετική θέση που θα βρίσκετε η βιβλιοθήκη ώστε να μπορέσει να βάλει την κατάλληλη τιμή μετά από την εντολή Call μέσα στον κώδικα. Για να μπορέσει να γίνει αυτό χρησιμοποιούνται δύο πίνακες ο **PLT(Procedure Linkage Table)** και ο **GOT(Global Offset Table)**.

Επειδή κάθε δυναμική βιβλιοθήκη πρέπει να μπορεί να φορτωθεί σε οποιαδήποτε περιοχή διευθύνσεων χρειάζεται να υπάρχει κάποιο σταθερό σημείο αναφοράς ώστε να μπορεί να ένα πρόγραμμα να χρησιμοποιήσει τις συναρτήσεις αυτής της βιβλιοθήκης. Το σταθερό αυτό σημείο αναφοράς είναι το **GOT(Global Offset Table)** που δεν είναι τυπωτά παραπάνω από ένα απλό πίνακα που οι έγραφες του είναι **διευθύνσεις μνήμης**. Οι διευθύνσεις αυτές μπορεί αν είναι εκεί που αρχίζουν στατικά δεδομένα της βιβλιοθήκες αλλά μπορεί να είναι και **που αρχίζει** η κάθε συνάρτηση της βιβλιοθήκης όταν αυτή έχει φορτωθεί στην μνήμη. Συγκεκριμένα ο **GOT** δεν περιέχει της **απόλυτες διευθύνσεις** αλλά της σχετικές θέσεις (Offset) **όπως θα συνέβαινε σε ένα Static Linking**. Το GOT table το παράγει ο Dynamic Linker και το τοποθετεί στον χώρο των δεδομένων της εφαρμογής αφού αυτή φορτωθεί στην μνήμη.

Όταν ένα πρόγραμμα φορτώνεται στην μνήμης τότε εκτός από το GOT έχει και το **PTL**. Το PLT είναι αυτό που θα περιέχει της διευθύνσεις μνήμης που βρίσκεται η κάθε μια συνάρτηση που υπάρχει στην βιβλιοθήκη. Ο πίνακας αυτός θα υπάρχει σε ένα πρόγραμμα άσχετα αν το πρόγραμμα θα χρησιμοποιήσει τις συναρτήσεις τής βιβλιοθήκης ή όχι. Το **PTL** προφανώς όταν πρώτοφορτώνεται δεν περιέχει τις διευθύνσεις των συναρτήσεων αλλά τα ονόματα τους. Τότε ο Dynamic Linker θα **κάνει Resolution** των ονομάτων με τις διευθύνσεις και θα γεμίσει τον πίνακα **PLT**. Ο πίνακα αυτός θα βρίσκεται στην text area και όχι στην Data area όπως το GOT.

Όταν λοιπών χρειάζεται να γίνει η κλήση μίας συναρτήσεις από την βιβλιοθήκη το πρόγραμμα είναι σχεδιασμένο να αναφέρεται στην κατάλληλη εγγράφη του πίνακα PLT. Ο πίνακας αυτός μετά το Dynamic Linking θα οδηγήσει τον EIP στην διεύθυνσης μνήμης που αρχίζει ο κώδικας αυτής της συνάρτησης. Τις περισσότερες φορές όμως να και ο GOT σχεδιαστικέ στο να αναφέρεται σε δεδομένα μίας δυναμικής βιβλιοθήκης αυτός χρησιμοποιείται και για αναφορά σε συναρτήσεις. Έτσι η συνηθισμένη διαδικασία είναι όταν γίνεται η κλίση μίας συνάρτησης από μία δυναμική βιβλιοθήκη πρώτα να γίνεται αναφορά στο PTL και από τον PTL με ένα Pointer προς συνάρτηση να γίνεται αναφορά στο **GOT. Τέλος από τον GOT ένας Pointer προς συνάρτηση οδηγεί στην επιθυμητή συνάρτηση**. Η διαδικασία φαίνεται στο παρακάτω σχήμα.



Σχήμα 15

Στο σχήμα αυτό φαίνεται ο έμμεσος τρόπος με τον οποίο καλείται μία συνάρτηση από μία δυναμικά συνδεόμενη βιβλιοθήκη. Σε κάθε πρόγραμμα που δημιουργείται στην C ο Compiler συνδέει δυναμικά την standard C βιβλιοθήκη. Αυτό το κάνει αφενός για να μπορούν να χρησιμοποιηθούν όλες οι συναρτήσεις αυτή της βιβλιοθήκης χωρίς να γίνει το συνηθισμένο #define αφετέρου γιατί αυτή η βιβλιοθήκη είναι κοινή και χρησιμοποιείται από όλα σχεδόν τα C προγράμματα. Συνεπώς έχοντας φορτωθεί αυτή η βιβλιοθήκη στην μνήμη σχεδόν από την εκκίνηση του λειτουργικού συστήματος το μόνο που χρειάζεται είναι να γίνει **μια δυναμική σύνδεση την βιβλιοθήκες** με την εκάστοτε εφαρμογή που θα την χρησιμοποιήσει.

4 Προγραμματισμός σε περιβάλλον 32bit Protected mode και Flat memory Addressing

Στα παλιά 16bit συστήματα όπως είπαμε ένα Segment δεν μπορούσε να είναι πάνω από 64KB αυτό είχε σαν αποτέλεσμα μεγάλα προγράμματα να αναγκάζονται να μοιράζουν τον κώδικα τους σε πολλά Segments. Αυτό είχε σαν συνέπεια ένα πρόγραμμα όταν έπρεπε να αλλάξει ένα ροή του προγράμματος και να την οδηγήσει σε μία συνάρτηση που βρισκόταν σε άλλο Segment να αλλάξει και το Segment. Για να συμβεί αυτό πρέπει μαζί τον EIP να αλλαχθεί ο αντίστοιχος Segment Register για παράδειγμα ο CS.

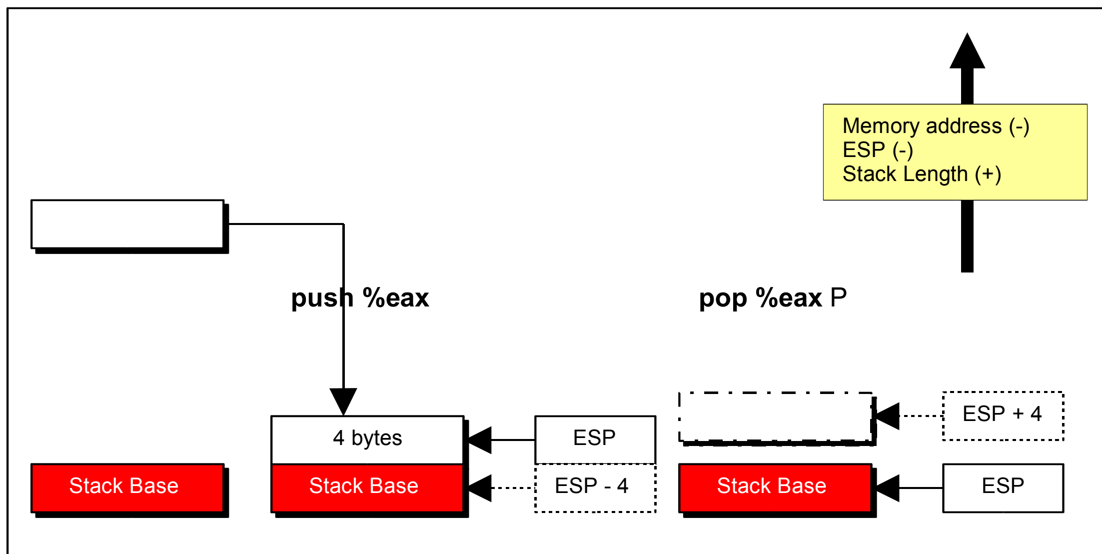
Από την εποχή που πρωτοεμφανίστικε το **Structural Programming** είναι γνωστό ότι για να γίνονται αυτές οι μεταβάσεις από μία συνάρτηση σε μία άλλη πρέπει οι τιμές των καταχωριτών να σωθούν κάπου στην μνήμη ακριβώς πριν την μετάβαση ώστε μετά από την χρήση μία συνάρτησης να επιστρέψει η ροή του προγράμματος στην κατάσταση που ήταν αρχικά. Για αυτό τον σκοπό ένα πρόγραμμα χρησιμοποιεί την στοίβα(Stack).

4.1 Η στοίβα και η χρήση της

Για τον χειρισμό της στοίβας στην 32 bit τεχνολογία όπως είπαμε χρησιμοποιούνται οι **SS** και **ESP registers**. Ακόμα υπάρχουν ο **EBX** που είναι καταχωριτής γενικής χρήσης αλλά και αυτός χρησιμοποιείται για την στοίβα πολλές φορές. Τέλος υπάρχει και ο **EBP(για Intell)** ή **FP** όπως λέγεται που και αυτός χρησιμοποιείται για την στοίβα και μάλιστα έχει ιδιαίτεροι σημασία για την εκτέλεση ενός προγράμματος.

Συγκεκριμένα ο **EBP (base pointer)** ή **FP(Frame pointer)** είναι αυτός που δείχνει το πού αρχίζει η περιοχή της στοίβας που χρησιμοποιείται από μία procedure/function. Ακόμα ο ρόλος του είναι να διαχωρίζει της παραμέτρους από τις τοπικές μεταβλητές μίας συνάρτησης. Για να καταλάβει κανείς την σημασία του στην ροή του προγράμματος αρκεί να γνωρίζει ότι το πρώτο πράγμα που σώζεται στην στοίβα αμέσως μετά την κλήση μίας συνάρτησης είναι η παλιά τιμή του **EBP**. Μετά το σώσιμο της παλαιάς τιμής παίρνει την τιμή που δείχνει εκείνη την στιγμή ο **ESP** και έτσι ορίζεται η αρχή του **Frame της συνάρτησης**.

Όπως αναφέραμε παραπάνω η διευθύνσεις στην στοίβα μειώνονται όσο ύψος της στοίβας ανεβαίνει (επειδή στη ουσία κατεβαίνει , βλέπε παραπάνω). Άρα αν θέλουμε να δεσμεύσουμε μνήμη στην στοίβα για αργότερα αρκεί να κάνουμε ένα **Subtract** στην στοίβα κατά 4 byte ή περισσότερο. Η τιμή των byte που θα δεσμευτούν στην στοίβα θα πρέπει να **διαιρείται ακριβός με το 4**. Στην στοίβα μπορούν να βάλουμε δεδομένα ή να βγάλουμε μονό σε μονάδες που διαιρούνται με το 4. Για να προσθέσουμε ή να αφαιρέσουμε δομένα στην στοίβα χρησιμοποιούμε τις εντολές **Push** και **Pop**. Για παράδειγμα όταν κάνουμε **pop** από τη στοίβα τότε αφαιρούνται τουλάχιστον 4 byte από την στοίβα και ο στον **ESP προστίθεται η τιμή 4**. Παράδειγμα της λειτουργίας της στοίβας φαίνεται παρακάτω.



Σχήμα 16

Η χρησιμότητα της στοίβας είναι ιδιαίτερα σημαντική όπως επισημάνθηκε από την αρχή για την υποστήριξη των **functions** και **procedures**. Γενικά είναι γνωστό ότι όταν γίνεται κλήση μίας συνάρτησης τότε το πρώτο πράγμα που θα γίνει να είναι **σωθεί στην στοίβα ο EIP** καταχωριτής. Ο **EIP** πριν την κλήση του προγράμματος έδειχνε στην επόμενη εντολή από αυτή της **εντολής κλίσης(Call)** της function. Η τιμή αυτή του **EIP** που σώζεται στην στοίβα ονομάζεται και **RET ή RA** και σημαίνει **Return Address**. Άμεσος μετά την κλίση της συνάρτησης θα πρέπει να σημαδευτή το κομμάτι της στοίβας που θα χρησιμοποιηθεί τις τοπικές μεταβλητές της. Ακόμα μπορεί να σωθεί και ο **EIP για την κλέιση μίας επόμενης συνάρτησης**.

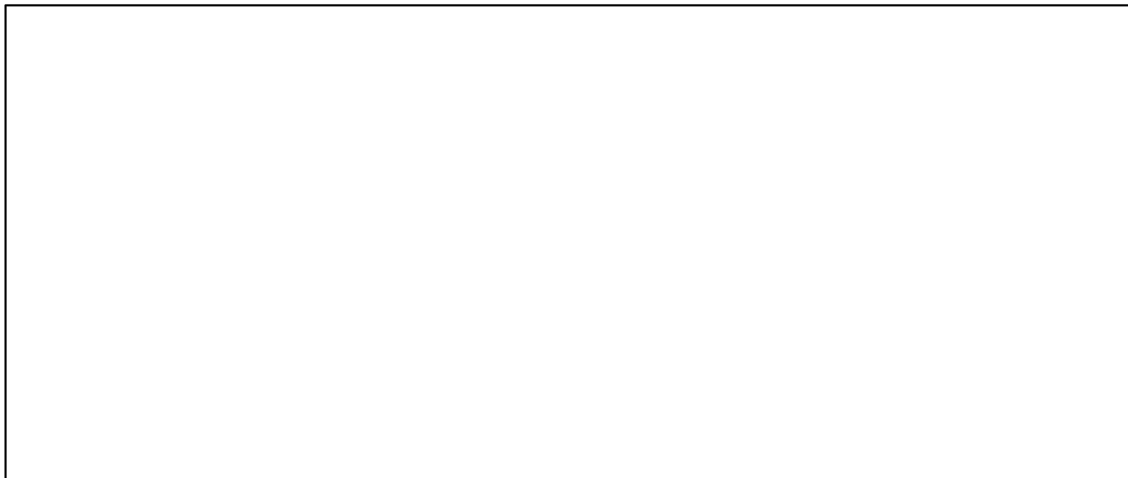
Εκτός όμως από τον **EIP** χρησιμοποιείται ο καταχωριστής **EBP** που αλλιώς λέγεται και FP που σκοπό έχει να σημαδέψει το frame στην στοίβα που θα χρησιμοποιηθεί από την συνάρτηση. Ο **EBP** χρησιμοποιείται σαν δείκτης της αρχή του frame επειδή ο **EIP** θα πρέπει να δείχνει πάντα τη κορυφή(τέλος) της στοίβας. Για να κρατηθεί όμως η αρχή της του Frame πρέπει να μεταφέρεται η τιμή το **ESP** στο **EBP** άμεσα μετά τη κλήση της συνάρτησης **και πριν την δέσμευση μνήμης** για τις τοπικές μεταβλητές. Για να μην χαθεί όμως τη τιμή του **EBP** που **δείχνει την αρχή του προηγούμενου frame στην στοίβα** σώζεται και η τιμή του **EBP** ακριβώς αφού έχει σωθεί ο **EIP.** Στο παρακάτω παράδειγμα φαίνεται αυτό ακριβός το που ειπώθηκε παραπάνω

Παράδιεγμα 1

```
                     /* C/C++ code */
void a_function(void) {

      char Buff[2] ; //a Buffer

}

main() {

      a_function(); //Calling function

}
```

```
                                /* Assembly Code */
.file   "test1.c"
        .version        "01.01"
gcc2_compiled.:
.text
        .align 4
.globl a_function
        .type    a_function,@function
a_function:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        leave
        ret
.Lfe1:
        .size    a_function,.Lfe1-a_function
        .align 4
.globl main
        .type    main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        call    a_function
        leave
    ret
.Lfe2:
        .size    main,.Lfe2-main
        .ident  "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.3 2.96-110)"
```

Όπως φαίνεται από το κώδικα σε Assembly αμέσως μετά την κλίση της συνάρτησης **a_function()** γίνεται push στην στοίβα ο EBP. Στην συνέχει αντιγράφεται ο ESP στον EBP. Φυσικά ο EIP έχει σωθεί στη στοίβα κατά την κλήση **a_fuction()** από την **main().** Η στοίβα μετά την κλίση της συνάρτησης διαμορφώνεται ως εξής:



Σχήμα 17

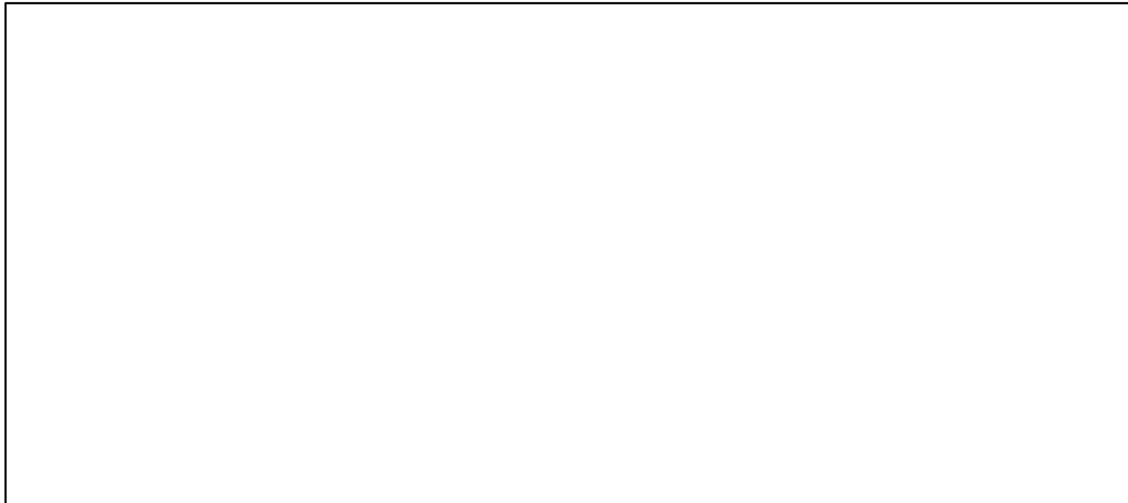
Στην περίπτωση που έχει παραμέτρους η συνάρτηση τότε οι παράμετροι μεταφέρονται μέσα στην στοίβα σε αντίθετη σειρά από αυτή που διαβάζονται από την συνάρτηση όπως φαίνεται στο *Παράδειγμα 2.2.*

Παράδειγμα 2

```
/* C/C++ code */
void a_function(char c) {

        char Buff[2] ; //a Buffer
        Buff[0]=c;
        Buff[1]=0x0; //end of string character

}

main() {
        char a;
        a='W';
        a_function(a); //Calling function

}
```

```
/* Assembly Code */
        .file   "test2.c"
         .version       "01.01"
gcc2_compiled.:
.text
         .align 4
.globl a_function
         .type    a_function,@function
a_function:
         pushl   %ebp
         movl    %esp, %ebp
         subl    $4, %esp
         movl    8(%ebp), %eax
         movb    %al, -1(%ebp)
         movb    -1(%ebp), %al
         movb    %al, -4(%ebp)
         movb    $0, -3(%ebp)
         leave
         ret
.Lfe1:
         .size    a_function,.Lfe1-a_function
         .align 4
.globl main
         .type    main,@function
main:
         pushl   %ebp
         movl    %esp, %ebp
         subl    $8, %esp
         movb    $87, -1(%ebp)
         subl    $12, %esp
         movsbl  -1(%ebp),%eax
         pushl   %eax
         call    a_function
         addl    $16, %esp
    leave
         ret
.Lfe2:
         .size    main,.Lfe2-main
         .ident  "GCC: (GNU) 2.96 20000731
```

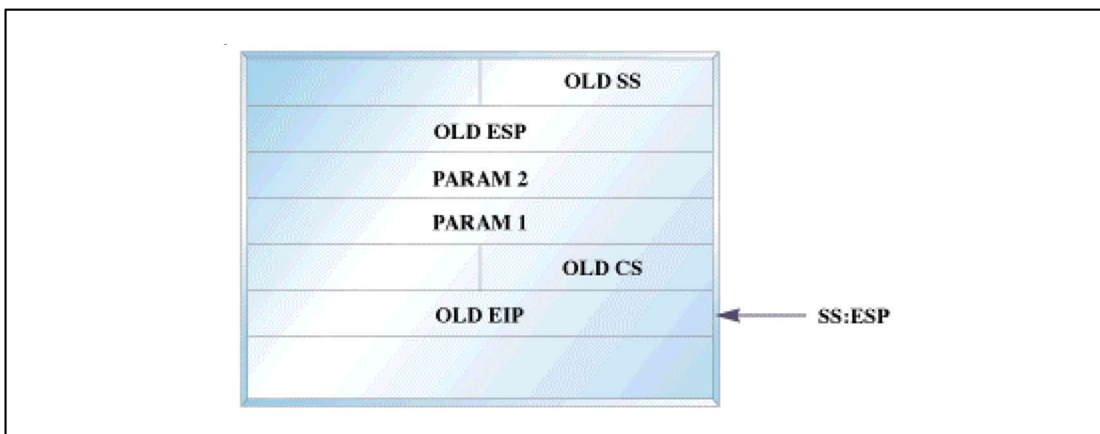
Η Στοίβα θα διαμορφωθεί ως εξής:



Σχήμα 18

Διαβάζοντας προσεχτικά τον κώδικα σε Assembly βλέπουμε πόσο χρήσιμος είναι ο καταχωριτής EBP. Παρατηρούμε ότι όλες οι τοπικές μεταβλητές καθώς και οι παράμετροι γίνονται προσβάσιμες με την **πρόσθεση ή αφαίρεση** της κατάλληλης τιμής στον EBP. Αυτή η προσθαφαίρεση λέγεται και **έμμεσης διευθησιοδώτησης** (**βλέπε memory addressing** στο κεφάλαιο 2).

Όλοι αυτή η διαδικασία που περιγράφετε παραπάνω συμβαίνει όταν η ένα πρόγραμμα έχει γίνει Compiled για **32 bit Protected mode** και μάλιστα όταν το λειτουργικό σύστημα χειρίζεται Segments των 4GB. Στην περίπτωση του **16bit Real mode** τα πράγματα είναι πιο σύνθετα όπως περιγράφεται από την αρχή του κεφαλαίου. Συγκεκριμένα η διάφορα της 16bit τεχνολογίας με την 32bit είναι σε δύο σημεία. Στην 16bit τεχνολογία η πρώτων ότι οι **μονάδες πληροφορίας** που σώζονται στην στοίβα είναι των 2 byte και όχι τον 4 byte όπως στην 32 bit. Η δεύτερη διαφορά είναι ότι στην 16 bit τεχνολογία χρειάζεται πολλές φορές εκτός από τον **Instruction Pointer** να σώζεται συχνά ο CS γιατί προκύπτει πολλές φορές η ανάγκη να αλλάζεται το Segment. Στην 32bit τεχνολογία αντίθετα δεν συμβαίνει αυτό παρά μόνο αν μια εφαρμογή χρειαστεί Segment μεγαλύτερο των **4GB**. Όταν συμβαίνει αυτό τότε όπως και στην 16bit έτσι και εδώ πρέπει **μαζί με τον EIP να σωθεί και ο CS** όπως φαίνεται στο παρακάτω σχήμα.



Σχήμα 19

Βιβλιογραφία

http://portal.acm.org/citation.cfm?id=173682.165159&coll=portal&dl=ACM&idx=J89&part=newsletter&WantType=newsletter&title=ACM%20SIGARCH%20Computer%20Architecture%20
News Citation

http://www.iecc.com/linker/linker10.html
Dynamic Linking and Loading

http://www.exposecorp.com/embedded/ex386.htm
Embedded 80386 Programming Examples-U

http://www.exposecorp.com/embedded/ex386.htm
Embedded 80386 Programming Examples

http://kernel.kaist.ac.kr/~jinsoo/course/cs330-2002spring/slides/supp-vm.pdf

http://www.amd.com/epd/processors/6.32bitproc/x21086/21086.pdf

http://www.netwinder.org/~scottb/notes/Elf-Design.html#toc4
NetWinder ELF Design Notes

http://www-106.ibm.com/developerworks/library/l-shobj/
Shared objects for the object disoriented

http://vx.netlux.org/texts/html/books/icz/tut4.html
Iczelion's Win32 Assembly Tutorial 4 Painting with Text

http://www.geocities.com/SiliconValley/Park/3230/x86asm/asmles00.html
Roby's PC Assembly Tutorial