ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
http://www.cslab.ece.ntua.gr

Λειτουργικά Συστήματα

7ο εξάμηνο, Ακαδημαϊκή περίοδος 2010-2011

Εξετάσεις Φεβρουαρίου 2011 – Λύσεις

Θέμα 1 (25%)

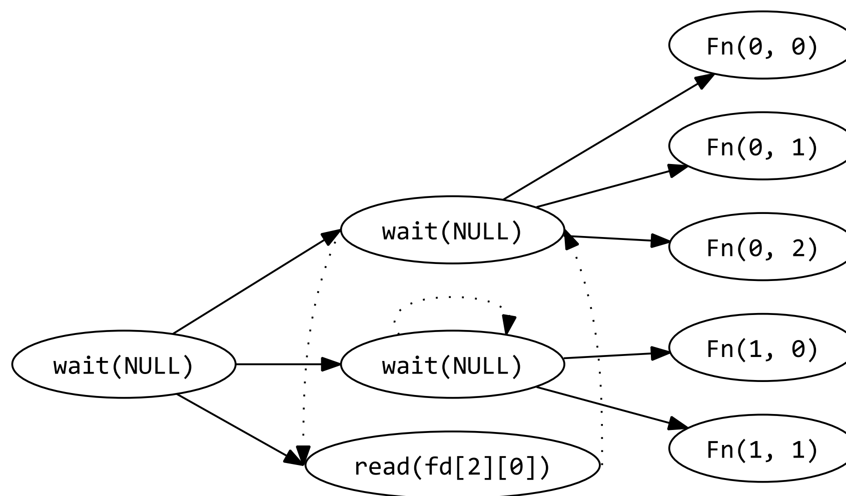
α. (20%) *Δίνεται το ακόλουθο τμήμα κώδικα:*

```
int fd[3][2], i, j, a, ret;

pipe(fd[0]); pipe(fd[1]); pipe(fd[2]);

for (i = 0; i < 3; i++) {
    ret = fork();
    if (ret != 0)
        continue;
    switch (i) {
        case 0:
            a = 3; write(fd[2][1], &a, sizeof(a)); break;
        case 1:
            a = 2; write(fd[1][1], &a, sizeof(a)); break;
        case 2:
            a = 4;
            read(fd[2][0], &a, sizeof(a)); write(fd[0][1], &a, sizeof(a));
    }
    read(fd[i][0], &a, sizeof(a));
    for (j = 0; j < a; j++) {
        ret = fork();
        if (ret == 0) {
            Fn(i, j); exit(0);
        }
    }
    for (j = 0; j < a; j++) wait(NULL);
}
```

```
for (i = 0; i < 3; i++) wait(NULL);

a = 1; write(fd[2][1], &a, sizeof(a));
```

Οι κλήσεις συστήματος δεν αποτυγχάνουν κι η `Fn()` δεν επιστρέφει ποτέ.
Σχεδιάστε το δέντρο διεργασιών που τελικά προκύπτει. Για κάθε κόμβο του δέντρου, γράψτε την κλήση συστήματος ή συνάρτηση που εκτελεί η αντίστοιχη διεργασία, μαζί με τα ορίσματά της. Συμπληρώστε το δέντρο διεργασιών ώστε να φαίνεται η διαδιεργασιακή επικοινωνία: για κάθε μεταφορά δεδομένων, σχεδιάστε ένα διακεκομμένο βέλος από τη διεργασία-αποστολέα στη διεργασία-παραλήπτη.

Προκύπτει το παρακάτω δέντρο διεργασιών:



Η 3η διεργασία-παιδί (`i = 2`) μπλοκάρει στην άδεια σωλήνωση `fd[2]` και δε γεννά ποτέ δικά της παιδιά.

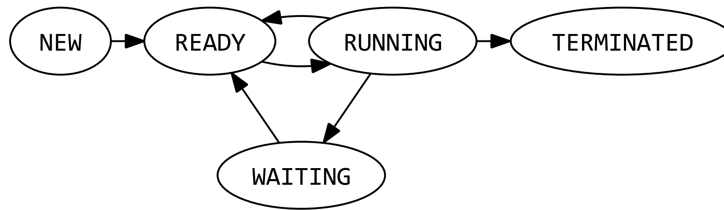
β. (5%) *Σχεδιάστε ένα ενδεικτικό διάγραμμα μετάβασης καταστάσεων διεργασίας σε σύγχρονο ΛΣ. Έστω οι παρακάτω δύο συναρτήσεις, `Fna()` και `Fnb()`:*

```
void Fna(void) {
    unsigned int i = 0;
    while (1) ++i;
}
```

```
void Fnb(void) {
    while (1) sleep(1);
}
```

Από ποιες καταστάσεις περνάει μια διεργασία όταν εκτελεί την `Fna()` κι από ποιες όταν εκτελεί την `Fnb()`; Ποιο γεγονός προκαλεί μετάβαση της διεργασίας από κατάσταση σε κατάσταση σε κάθε περίπτωση; Υποθέστε ότι υπάρχουν κι άλλες διεργασίες στο σύστημα κι ο αλγόριθμος χρονοδρομολόγησης είναι Round-Robin.

Ένα ενδεικτικό διάγραμμα μετάβασης καταστάσεων διεργασίας παρουσιάζεται στο ακόλουθο σχήμα.



Μια διεργασία που εκτελεί την `Fna()` γίνεται READY $\longrightarrow$ RUNNING και RUNNING $\longrightarrow$ READY. Το πρώτο συμβαίνει όταν επιλέγεται από τον χρονοδρομολογητή προς εκτέλεση, το δεύτερο όταν εκπνεύσει το κβάντο χρόνου της. Μια διεργασία που εκτελεί την `Fnb()` γίνεται READY $\longrightarrow$ RUNNING, RUNNING $\longrightarrow$ WAITING και WAITING $\longrightarrow$ READY. Η πρώτη μετάβαση συμβαίνει όταν επιλεγεί από τον χρονοδρομολογητή, η δεύτερη όταν εκτελέσει την κλήση συστήματος `sleep()` και η τρίτη όταν συμβεί διακοπή χρονιστή και το ΛΣ δει ότι έχει περάσει $1s$ στην κατάσταση WAITING οπότε πρέπει να ξαναγίνει READY.

Θέμα 2 (25%)

Δίνονται λειτουργίες δομής πίνακα κατακερματισμού (hash table), που αποθηκεύει σύνολο κλειδιών. Η δομή υλοποιείται με πίνακα `N` θέσεων. Σε κάθε θέση είναι αποθηκευμένη μια λίστα. Ο κάθε κόμβος της λίστας περιέχει ένα κλειδί (key).

```
list table[N];
```

```
void insert(int key){
  list a;
  a = table[key % N];
  append(a, key);
}
```

```
void remove(int key){
  list a;
  a = table[key % N];
  delete(a, key);
}
```

```
int exists(int key){
  list a;
  a = table[key % N];
  return contains(a, key);
}
```

Σημείωση #1*: Οι κλήσεις χειρισμού της λίστας:*

- `append()`*: προσθήκη στοιχείου στη λίστα*
- `delete()`*: διαγραφή στοιχείου από τη λίστα*
- `contains()`*: έλεγχος αν ένα στοιχείο υπάρχει στη λίστα*

***ΔΕΝ** περιέχουν κάποιου είδους συγχρονισμό. Σε περίπτωση αμφιβολίας για το πώς υλοποιούνται καλείστε να είστε συντηρητικοί!*

Σημείωση #2*: Για απλότητα θεωρούμε ότι δεν υπάρχει περίπτωση η* `insert()` *να κληθεί με τιμή κλειδιού που υπάρχει ήδη στον πίνακα και ότι δεν υπάρχει περίπτωση η* `remove()` *να κληθεί με τιμή κλειδιού που δεν υπάρχει στον πίνακα.*

***α.** (5%) Υλοποιήστε σχήμα συγχρονισμού για (όλες) τις παραπάνω λειτουργίες χρησιμοποιώντας κεντρικό κλείδωμα.*

Με κεντρικό κλείδωμα οι λειτουργίες `insert`, `remove`, `exists` διαμορφώνονται ως εξής:

```
list table[N];
semaphore biglock;
```

```
void insert(int key){
  list a;
  a = table[key % N];
  wait(biglock);
  append(a, key);
  signal(biglock);
}
```

```
void remove(int key){
  list a;
  a = table[key % N];
  wait(biglock);
  delete(a, key);
  signal(biglock);
}
```

```
int exists(int key){
  list a;
  int ret;
  a = table[key % N];
  wait(biglock);
  ret = contains(a, key);
  signal(biglock);
  return ret;
}
```

β. (10%) *Ποιό είναι το μειονέκτημα του κεντρικού κλειδώματος; Υλοποιήστε σχήμα συγχρονισμού πολλαπλών κλειδωμάτων που να το αντιμετωπίζει.*

Το μειονέκτημα του κεντρικού κλειδώματος είναι ότι κλειδώνει το σύνολο των λιστών που χρησιμοποιεί το hash table, παρόλο που κάθε λειτουργία χρειάζεται πρόσβαση σε μόνο μία λίστα, αυτή στην οποία ανήκει το `key`. Για να το αντιμετωπίσουμε θα υλοποιήσουμε fine-grained locking, με ένα κλείδωμα ανά λίστα, ως εξής:

```
list table[N];
semaphore locks[N];
```

```
void insert(int key){
  list a;
  a = table[key % N];
  wait(locks[key % N]);
  append(a, key);
  signal(locks[key % N]);
}
```

```
void remove(int key){
  list a;
  a = table[key % N];
  wait(locks[key % N]);
  delete(a, key);
  signal(locks[key % N]);
}
```

```
int exists(int key){
  list a;
  int ret;
  a = table[key % N];
  wait(locks[key % N]);
  ret = contains(a, key);
  signal(locks[key % N]);
  return ret;
}
```

γ. (10%) *Στο σχήμα συγχρονισμού του ερωτήματος **β** ζητείται να υλοποιηθεί συνάρτηση `insertremove(key1,key2)` που* ατομικά *εισάγει το `key1` και διαγράφει το `key2`.*
Σωστή ατομική υλοποίηση της παραπάνω λειτουργίας συνεπάγεται ότι, δεδομένου ότι το `key2` υπάρχει στη δομή, η έκφραση `(!exists(key2)) && (!exists(key1))` δεν επιστρέφει ποτέ *`true`.* ΠΡΟΣΟΧΗ: *Βεβαιωθείτε ότι η υλοποίησή σας δεν εμφανίζει deadlocks.*

Η `insertremove()` χρειάζεται να κλειδώνει *δύο* λίστες, μία για το `key1` και μία για το `key2`. Για να αποφευχθεί το deadlock, χρειάζεται ειδικός χειρισμός των εξής σημείων:

- Αν τα `key1`, `key2` ανήκουν στην ίδια λίστα, πρέπει να προσπαθήσει μόνο μία φορά να αποκτήσει το αντίστοιχο κλείδωμα.
- Πρέπει να αποκτά τα κλειδώματα με `συγκεκριμένη` σειρά, για να αποφύγει deadlock με μια κλήση `insertremove(key2, key1)`.

Οπότε:

```
void insertremove(int key1, int key2) {
  list a1, a2;
  int first, second;

  k1 = key1 % N; k2 = key2 % N;
  a1 = table[k1]; a2 = table[k2];

  if (k1 < k2) {
    first = k1; second = k2;
  } else {
    first = k2; second = k1;
  }

  wait(locks[first]);
  if (second != first)
    wait(locks[second]);
  insert(a1, key1);
  remove(a2, key2);
  if (second != first)
    signal(locks[second]);
  signal(locks[first]);
}
```

Θέμα 3 (20%)

α. (10%) *Έστω σύστημα με περισσότερους του ενός επεξεργαστές. Η χρονοδρομολόγηση υλοποιείται με μια ουρά χρονοδρομολόγησης ανά επεξεργαστή. Για ευκολία θεωρούμε ότι αρχικά κάθε ουρά περιέχει μια διεργασία.*
Έστω δύο τακτικές χρονοδρομολόγησης:

α′ Κάθε διεργασία που δημιουργείται παραμένει μέχρι την ολοκλήρωσή της στην ουρά, στην οποία άνηκε ο γονιός της.

β′ Κατά την δημιουργία μιας διεργασίας ο χρονοδρομολογητής της ουράς του γονιού της ελέγχει όλες τις υπόλοιπες ουρές, και την τοποθετεί σε αυτή με το μικρότερο αριθμό διεργασιών.

Αναφέρατε ένα πλεονέκτημα της τακτικής (α’) έναντι της (β’), και ένα πλεονέκτημα της τακτικής (β’) έναντι της (α’). Μπορείτε, αν κρίνετε σκόπιμο, να επικαλεστείτε παραδείγματα.

Ένα πλεονέκτημα της τακτικής (α’) έναντι της (β’) είναι ότι αν ένας χρήστης αποφασίσει να δημιουργήσει πάρα πολλές διεργασίες-παιδιά, αυτές παραμένουν όλες στην ίδια ουρά κι ανταγωνίζονται για χρόνο σε έναν επεξεργαστή, οπότε δεν μπορεί ένας χρήστης να προκαλέσει υπερβολικό φόρτο στο σύστημα και μειωμένη διαθεσιμότητα της CPU για διεργασίες άλλων χρηστών.
Αντίστροφα, ένα πλεονέκτημα της (β’) έναντι της (α’) είναι ότι τα παιδιά ανατίθενται προς εκτέλεση στους υπόλοιπους επεξεργαστές οπότε γίνεται καλύτερη αξιοποίηση του υλικού και δεν μένουν άεργοι επεξεργαστές.
Από την άλλη πλευρά, η (β’) έχει το μειονέκτημα ότι κάθε χρονοδρομολογητής δεν μπορεί να αποφασίσει μόνο με τοπικά κριτήρια, αλλά χρειάζεται να ελέγχει ουρές άλλων

επεξεργαστών, οπότε αυξάνεται το κόστος της χρονοδρομολόγησης. Επιπλέον, συχνά ο πατέρας επικοινωνεί με τα παιδιά του, π.χ. μέσω μοιραζόμενης μνήμης. Μετακίνηση των διεργασιών σε άλλους επεξεργαστές συνεπάγεται μετακίνηση δεδομένων μέσω του δικτύου διασύνδεσής τους (π.χ. δίαυλος συστήματος), αντί για ανταλλαγή δεδομένων μέσω της κρυφής μνήμης του επεξεργαστή.

β. (10%)

i. Αναφέρατε ένα πλεονέκτημα κι ένα μειονέκτημα της χρήσης μεγάλου μεγέθους μπλοκ δεδομένων σε ένα σύστημα αρχείων.

Χρήση μεγάλου μέγεθος μπλοκ δεδομένων οδηγεί σε καλύτερη επίδοση, γιατί τα περιεχόμενα του αρχείου παραμένουν σε κοντινές περιοχές του δίσκου, οπότε έρχονται με μία λειτουργία Ε/Ε στη μνήμη όταν χρειάζεται να ανακτηθούν (χωρική τοπικότητα). Επιπλέον, μειώνεται το διαχειριστικό κόστος, λόγω μικρότερου αριθμού μπλοκ για δεδομένη χωρητικότητα δίσκου.
Από την άλλη πλευρά, χρήση μεγάλου μεγέθους μπλοκ επιβαρύνει τον εσωτερικό κατακερματισμό και αυξάνει το ποσοστό της χωρητικότητας του δίσκου που χάνεται γιατί το μέγεθος των αρχείων δεν είναι ακριβές πολλαπλάσιο του μεγέθους του μπλοκ – ο δίσκος μπορεί να είναι γεμάτος πολλά πολύ μικρά αρχεία.

ii. Θεωρούμε ότι το FCB (i-node) ενός συστήματος αρχείων περιέχει ένα μετρητή (ακέραιο αριθμό) για την υλοποίηση μη συμβολικών *συνδέσμων (hard links). Περιγράψτε συνοπτικά τι συνεπάγεται η εκτέλεση των παρακάτω λειτουργιών για τις δομές i-node που υπάρχουν και τις τιμές των μετρητών που περιέχουν.*

- *Δημιουργία νέου αρχείου*
- *Δημιουργία νέου hard link προς υπάρχον αρχείο*
- *Διαγραφή αρχείου (π.χ. `rm file1`).*

Ο μετρητής ενός i-node καταγράφει πόσα hard links υπάρχουν προς το αρχείο αυτό. Κατά τη δημιουργία ενός αρχείου, κατασκευάζεται μια νέα δομή i-node κι ο μετρητής της τίθεται στην τιμή `1`. Κατά τη δημιουργία νέου hard link σε υπάρχον αρχείο, δεν κατασκευάζεται νέα δομή i-node, αλλά ο μετρητής της υπάρχουσας δομής i-node αυξάνεται κατά 1. Κατά τη διαγραφή ενός αρχείου, π.χ. του `file1` ο μετρητής της αντίστοιχης δομής i-node μειώνεται κατά 1. Αν φτάσει την τιμή `0` τότε απελευθερώνεται και η δομή i-node.

Θέμα 4 (30%)

α. (15%) *Μια διεργασία διαθέτει τρία πλαίσια φυσικής μνήμης κι εκτελεί την παρακάτω ακολουθία αναφορών σε σελίδες εικονικής μνήμης:*

1, 3, 2, 1, 4, 2, 5, 3, 2, 1, 5

Και τα τρία πλαίσια είναι αρχικά άδεια. Εκτελέστε τον αλγόριθμο αντικατάστασης σελίδας και βρείτε τον αριθμό των σφαλμάτων σελίδας που συμβαίνουν όταν η στρατηγική είναι:

i. First-In, First-Out (FIFO)

ii. Least Recently Used (LRU)

	1	3	2	1	4	2	5	3	2	1	5	
FIFO	1	1 3	1 3 2		4 3 2		4 5 2	4 5 3	2 5 3	2 1 3	2 1 5	9 faults
LRU	1	1 3	1 3 2		1 4 2		5 4 2	5 3 2		1 3 2	1 5 2	8 faults

Ένας συμφοιτητής σας ισχυρίζεται ότι έχει βρει στρατηγική αντικατάστασης σελίδων η οποία εμφανίζει 5 σφάλματα σελίδας για τη συγκεκριμένη ακολουθία. Τι του απαντάτε;
Ο συμφοιτητής μας ισχυρίζεται ότι έχει βρει στρατηγική με 5 σφάλματα, όμως με βέλτιστη αντικατάσταση έχουμε 6 σφάλματα:

	1	3	2	1	4	2	5	3	2	1	5	
OPT	1	1 3	1 3 2		4 3 2		5 3 2			5 1 2		6 faults

άρα θα του πούμε ότι δεν πρέπει να επιμένει, γιατί κάνει λάθος.
Αλλιώς: Η ακολουθία αναφέρεται σε 5 ξεχωριστές σελίδες, άρα αναγκαστικά έχουμε 5 misses (compulsory). Μετά την πρώτη αναφορά στην 5η σελίδα, θα βρίσκονται 3 σελίδες στη φυσική μνήμη. Όμως, η ακολουθία έχει ακόμη 4 αναφορές σε διακριτές σελίδες (3, 2, 1, 5), άρα όποιες 3 σελίδες και να είναι εκείνη τη στιγμή στη φυσική μνήμη, ένα page fault θα γίνει. Οπότε η ακολουθία προκαλεί τουλάχιστον 6 σφάλματα.

β. (5%) *Η υλοποίηση αλγορίθμων αντικατάστασης σελίδας με προσέγγιση LRU διευκολύνεται σημαντικά όταν η μηχανή διαθέτει bit αναφοράς (reference bit) για κάθε σελίδα εικονικής μνήμης. Έστω μηχανή η οποία δεν διαθέτει αυτή τη δυνατότητα. Σκιαγραφήστε στρατηγική με την οποία το ΛΣ μπορεί να προσομοιώσει τη λειτουργικότητα του reference bit και να γνωρίζει σε ποιες σελίδες έγινε αναφορά μέσα σε ένα συγκεκριμένο χρονικό διάστημα.*
Χρειαζόμαστε έναν τρόπο να μπορεί το ΛΣ να εντοπίσει σε ποιες σελίδες έγινε αναφορά, στο χρονικό διάστημα από t_0 έως t_1. Όταν είχαμε reference bits υποστηριζόμενα από το υλικό, το *υλικό* φρόντιζε να ανάβει το reference bit για τις σελίδες στις οποίες γινόταν αναφορά με εντολές load ή store. Πλέον χρειάζεται το ΛΣ να μπορεί να εντοπίσει σε ποιες σελίδες αναφέρθηκε μια διεργασία *χωρίς* υποστήριξη από το υλικό.
Μια λύση είναι τη χρονική στιγμη t_0 το ΛΣ να *ακυρώνει* όλες τις εγγραφές του πίνακα σελίδων μιας διεργασίας. Κάθε πρώτη αναφορά σε σελίδα προκαλεί *σφάλμα* σελίδας, οπότε ξυπνάει το ΛΣ, θέτει στη σωστή τιμή την εγγραφή του πίνακα σελίδων και καταγράφει ότι στη συγκεκριμένη σελίδα έγινε αναφορά. Οι επόμενες αναφορές στη σελίδα δεν προκαλούν page fault. Τη χρονική στιγμή t_1 το ΛΣ ξέρει σε ποιες σελίδες έγινε αναφορά – είναι αυτές που έχουν έγκυρη απεικόνιση στον πίνακα σελίδων – και μπορεί να ακυρώσει όλες τις εγγραφές για να ξεκινήσει εκ νέου τη διαδικασία για το επόμενο χρονικό διάστημα.

γ. (10%) *Αναφέρατε πέντε χαρακτηριστικά ενός σύγχρονου Λειτουργικού Συστήματος τα οποία είτε βασίζονται για την υλοποίησή τους είτε επωφελούνται από συγκεκριμένους μηχανισμούς του Υλικού. Για κάθε ένα από τα χαρακτηριστικά αυτά μιλήστε περιληπτικά, με μία πρόταση, για τον τρόπο με τον οποίο υποστηρίζονται από τον αντίστοιχο μηχανισμό του Υλικού.*
Ενδεικτικά:

- **ΛΣ:** Χρονοδρομολόγηση **Υλικό:** Χρονιστής (timer)
  Πριν από μεταγωγή περιεχομένου (context switch) ο χρονοδρομολογητής θέτει τον χρονιστή ώστε να προκληθεί διακοπή υλικού, η οποία σταματά την εκτέλεση της τρέχουσας διεργασίας όταν εκπνεύσει το κβάντο χρόνου.

- **ΛΣ:** Αποδοτικός χειρισμός συσκευών Ε/Ε **Υλικό:** Διακοπές Υλικού
  Το υλικό επιτρέπει στο ΛΣ να διαθέτει τη CPU για εκτέλεση κώδικα ενώ μια εξωτερική συσκευή εργάζεται παράλληλα. Όταν η συσκευή ολοκληρώσει τη δουλειά που της ανατέθηκε, προκαλεί μια διακοπή υλικού, η οποία προκαλεί την εκτέλεση ρουτίνας εξυπηρέτησης του ΛΣ.

- **ΛΣ:** Εικονική μνήμη **Υλικό:** MMU - Μετάφραση διευθύνσεων με σελιδοποίηση
  Το υλικό μπορεί να κάνει μετάφραση διευθύνσεων με σελιδοποίηση και διαθέτει ειδική μονάδα διαχείριση μνήμης (MMU).

- **ΛΣ:** Προστασία - Καταστάσεις πυρήνα/χρήστη **Υλικό:** Προνομιούχες εντολές - mode bit
  Το υλικό ξέρει αν εκτελεί κώδικα σε κατάσταση πυρήνα ή σε κατάσταση χρήστη (με βάση την τιμή του mode bit). Σε κατάσταση χρήστη κλήση μιας προνομιούχου εντολής προκαλεί *trap*, οπότε ο επεξεργαστής περνάει σε κατάσταση πυρήνα κι εκτελεί προκαθορισμένη ρουτίνα εξυπηρέτησης διακοπής του ΛΣ.

- **ΛΣ:** Προσέγγιση LRU **Υλικό:** reference bit
  Κάθε σελίδα στον πίνακα σελίδων διαθέτει reference bit με το οποίο το ΛΣ μπορεί να εντοπίζει σε ποιες σελίδες έγινε πρόσβαση από την τελευταία φορά που μηδένισε τα αντίστοιχα reference bits.

- **ΛΣ:** Αποδοτική σελιδοποίηση **Υλικό:** modify bit
  Το υλικό μαρκάρει μια σελίδα ως βρώμικη, ανάβοντας το modify bit όταν αλλάζει το περιεχόμενό της, ώστε το ΛΣ να γνωρίζει ότι αυτή η σελίδα έχει αλλάξει και χρειάζεται να την αντιγράψει στο δίσκο πριν απελευθερώσει το πλαίσιο στο οποίο είναι αποθηκευμένη.

- **ΛΣ:** Προστασία μνήμης - υλοποίηση CoW κ.ά. **Υλικό:** bits προστασίας στις σελίδες
  Κάθε σελίδα μνήμης διαθέτει bits προστασίας `r,w,x` ώστε το ΛΣ να μπορεί να ρυθμίζει με ακρίβεια τα δικαιώματα πρόσβασης.

- **ΛΣ:** Συγχρονισμός - κρίσιμα τμήματα **Υλικό:** Ατομικές εντολές
  Το υλικό διαθέτει εντολές – π.χ. Enable/disable Interrupts, Test-and-Set, Swap – χρήσιμες για υλοποίηση μηχανισμών συγχρονισμού.