Πανεπιστήμιο Μακεδονίας
Τμήμα Εφαρμοσμένης Πληροφορικής

Συνοπτικές Σημειώσεις

για το μάθημα

# Λειτουργικά Συστήματα

Μάνος Ρουμελιώτης

Καθηγητής

Θεσσαλονίκη, Μάιος 2008

# 1 Εισαγωγή

Το λειτουργικό σύστημα είναι το σύνολο των προγραμμάτων ενός υπολογιστικού συστήματος πού καλύπτει τον έλεγχο των διάφορων συσκευών του συστήματος καθώς και την επικοινωνία του χρήστη με το σύστημα. Τα προγράμματα αυτά προσπαθούν να κάνουν πιο αποδοτικό το σύστημα, να απλοποιήσουν τη χρήση του, και να βελτιστοποιήσουν την απόδοσή του. Αν δούμε το λειτουργικό σύστημα σαν διαχειριστή των υποσυστημάτων του συστήματος, τότε θα πρέπει να εκτελεί τις εξής λειτουργίες:

1. Επιβλέπει τη χρήση των υποσυστημάτων
2. Καθορίζει ποιός παίρνει τι, πότε, και πόσο
3. Απονέμει το υποσύστημα
4. Επανακτά το υποσύστημα

Τα κυριώτερα υποσυστήματα πού ελέγχονται από το λειτουργικό σύστημα είναι: Η κυρίως μνήμη, η CPU, οι μονάδες εισόδου/εξόδου, και οι πληροφορίες.

## 1.1 Λειτουργίες διαχείρισης της μνήμης

1. Επιβλέπει τη χρήση της κυρίως μνήμης. Ποιά τμήματα χρησιμοποιούνται και από ποιόν, ποιά τμήματα είναι ελεύθερα (free).
2. Αν το σύστημα χρησιμοποιεί multiprogramming[1], αποφασίζει για το ποιά διεργασία παίρνει μνήμη, πότε θα την πάρει, και πόση θα πάρει.
3. Απονέμει το υποσύστημα (μνήμη) όταν η διεργασία την ζητήσει και η διαδικασία 2 το επιτρέπει.
4. Επανακτά τη μνήμη όταν η διεργασία δεν τη χρειάζεται πλέον.

## 1.2 Λειτουργίες διαχείρισης του processor

1. Επιβλέπει τη χρήση του ή των processor(s). Το πρόγραμμα που το κάνει αυτό λέγεται (traffic controller).
2. Αποφασίζει για το ποιός θα χρησιμοποιήσει τον processor. Το πρόγραμμα που το κάνει αυτό λέγεται processor scheduler.
3. Απονέμει το υποσύστημα (processor) τροποποιώντας τα περιεχόμενα κάποιων καταχωρητών (registers) στο hardware. Αυτό λέγεται συνήθως dispatcher.
4. Επανακτά τον processor όταν η process τερματιστεί, απελευθερώσει τον processor, ή υπερβεί τον επιτρεπόμενο χρόνο χρήσης.

## 1.3 Λειτουργίες διαχείρισης περιφερειακών μονάδων

[1]Ο όρος Multiprogramming χρησιμοποιείται για να υποδηλώσει την ταυτόχρονη "εκτέλεση" πολλών προγραμμάτων. Αυτό σημαίνει ότι πολλά προγράμματα είναι στην κατάσταση εκτέλεσης, χωρίς όμως να σημαίνει ότι η CPU τα εκτελεί ταυτόχρονα. Απλώς είτε βρίσκονται στην κυρίως μνήμη περιμένοντας τη σειρά τους για εκτέλεση, είτε περιμένουν τον τερματισμό κάποιας λειτουργίας εισόδου/εξόδου. Μόνο ένα πρόγραμμα εκτελείται πραγματικά από την CPU κάποια δεδομένη χρονική στιγμή.

1. Επιβλέπει τη χρήση των μονάδων εισόδου/εξόδου. Αυτό λέγεται συνήθως I/O traffic controller.
2. Αποφασίζει για το ποιά process θα χρησιμοποιήσει κάποια μονάδα, και για πόσο διάστημα. Αυτό λέγεται I/O scheduler.
3. Απονέμει το περιφερειακό και το ενεργοποιεί.
4. Επανακτά το περιφερειακό όταν η process δεν το χρειάζεται πλέον.

## 1.4 Λειτουργίες διαχείρισης της πληροφορίας

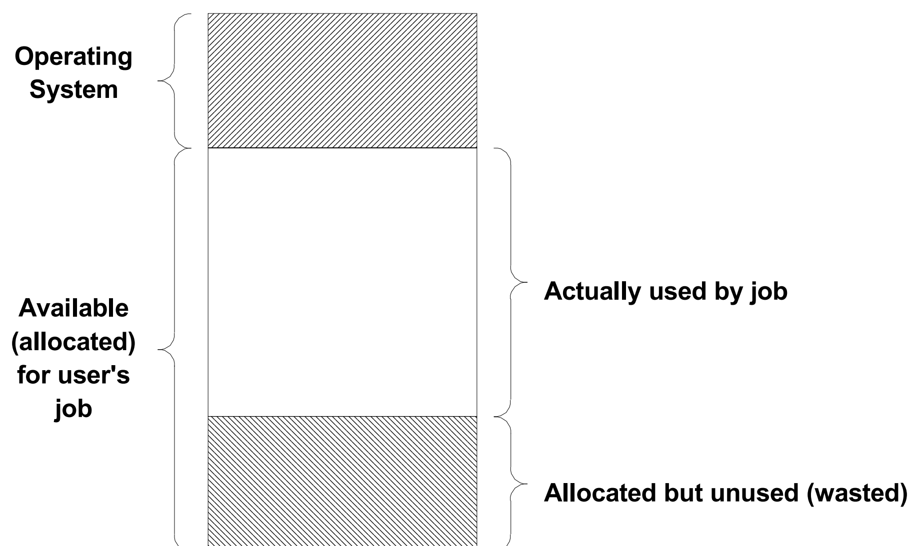
1. Επιβλέπει τη χρήση της πληροφορίας, την θέση της, τη χρήση της κλπ. Το σύνολο των λειτουργιών αυτών λέγεται file system.
2. Αποφασίζει για το ποιός θα χρησιμοποιήσει την πληροφορία, επιβάλει τις απαιτήσεις προστασίας, και διαθέτει τις υπορρουτίνες προσπέλασης.
3. Απονέμει την πληροφορία όταν η process τη ζητήσει. (δηλ. open file).
4. Επανακτά την πληροφορία όταν η process δεν τη χρειάζεται πλέον (δηλ. close file).

# 2 Λειτουργίες διαχείρισης της μνήμης

Η διαχείριση της μνήμης από το λειτουργικό σύστημα εξαρτάται από πολλούς παράγοντες: πόση μνήμη έχει το σύστημα, αν το χρησιμοποιούν ένας ή πολλοί χρήστες, αν κάνει multiprogramming ή όχι κλπ. Θα εξετάσουμε τους δύο κυριώτερους τρόπους διαχείρισης της μνήμης, single contiguous allocation, και partitioned allocation.

## 2.1 Single contiguous allocation

Η μέθοδος αυτή είναι πολύ απλή και δε χρειάζεται συγκεκριμένα χαρακτηριστικά από το hardware. Χρησιμοποιείται κυρίως σε μικρά συστήματα όπως το IBM PC που δεν χρησιμοποιούν multiprogramming. Η μνήμη απονέμεται σε ένα μόνο job, όπως φαίνεται στο Σχήμα 2.1.



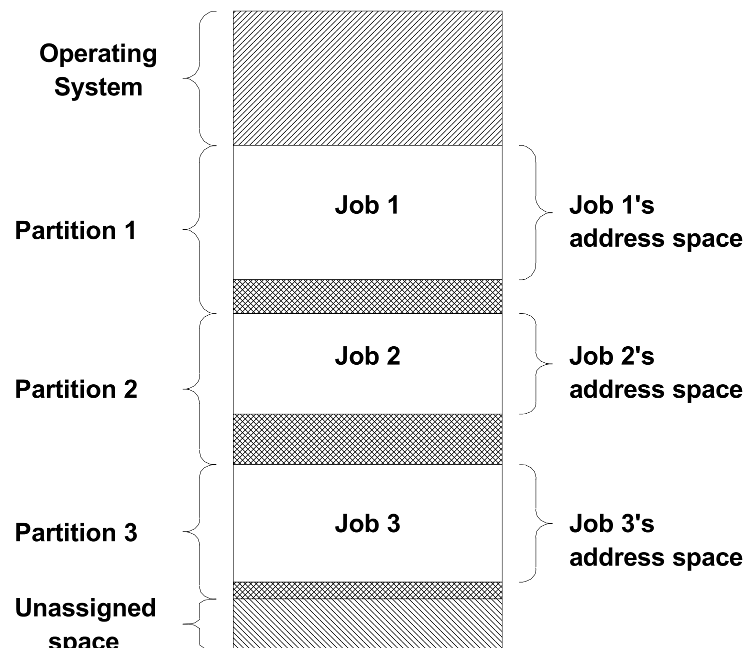

Σχήμα 2.1 Single contiguous allocation

Στην ουσία η μνήμη χωρίζεται σε τρία συνεχή τμήματα. Ένα μέρος της μνήμης ανήκει στο λειτουργικό σύστημα και χρησιμοποιείται συνεχώς απ' αυτό. Η υπόλοιπη μνήμη είναι

διαθέσιμη και απονέμεται στο μοναδικό job που εκτελείται από το σύστημα. Αυτό το job μπορεί να μην χρησιμοποιήσει όλη τη διαθέσιμη μνήμη, στην οποία περίπτωση το υπόλοιπο της μνήμης μένει αχρησιμοποίητο.

## 2.2 Partitioned allocation

Όταν το σύστημα χρησιμοποιεί multiprogramming, όλα τα jobs πού εκτελούνται από την CPU θα πρέπει να βρίσκονται στη μνήμη. Το λειτουργικό σύστημα χωρίζει τη μνήμη σε τμήματα (partitions) και βάζει το κάθε job σε διαφορετικό τμήμα (Σχήμα 2.2)




Σχήμα 2.2 Partitioned allocation

Υπάρχουν διάφοροι τρόποι να καθορισθούν τα τμήματα της μνήμης πού απαιτούνται για κάθε job, καθώς επίσης και το ποιά jobs θα μπουν στη μνήμη για εκτέλεση.

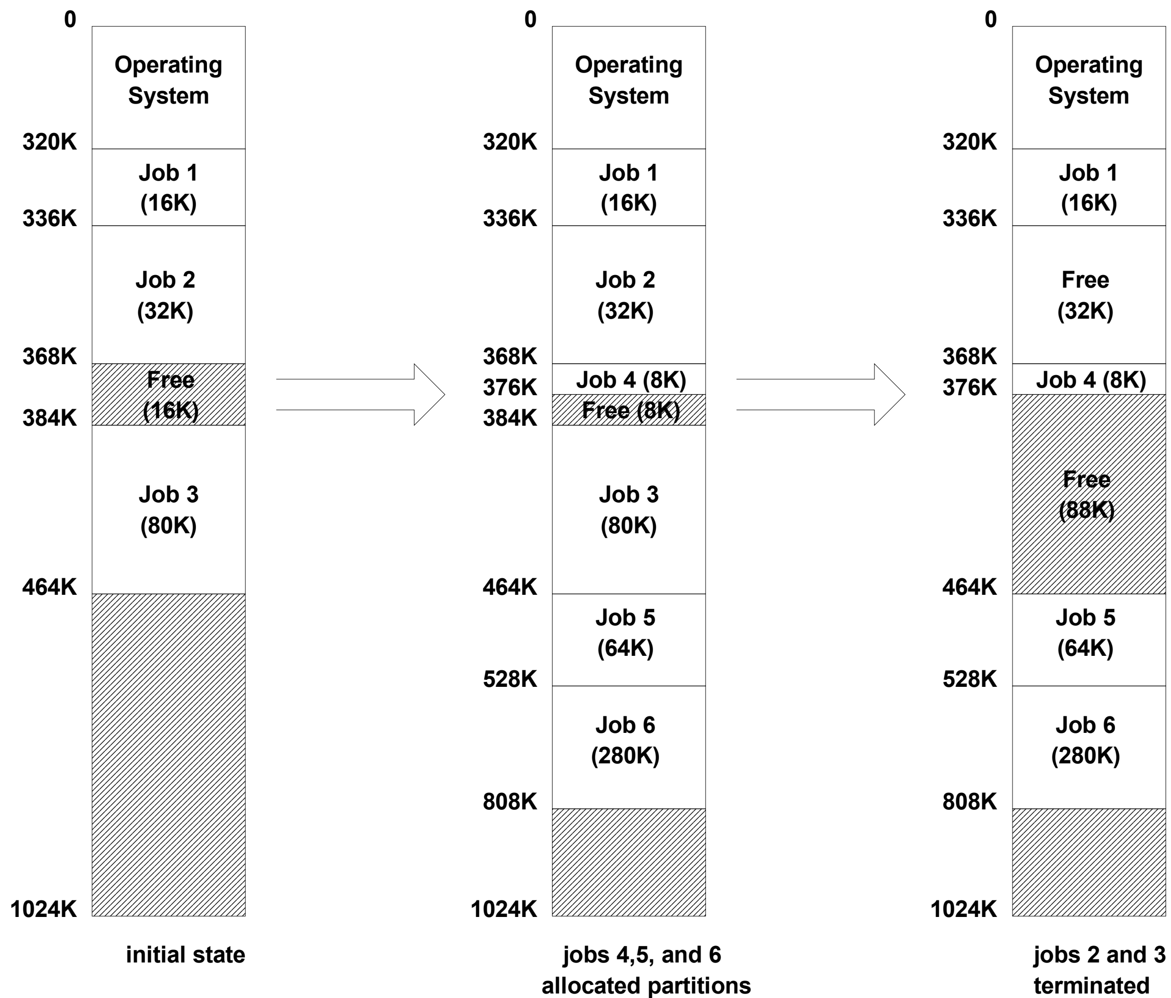
### 2.2.1 Static Allocation

Κατά την στατική απονομή της μνήμης, το λειτουργικό σύστημα ελέγχει τα jobs πού περιμένουν για εκτέλεση και καθορίζει την ποσότητα της μνήμης που χρειάζεται το καθένα. Κατόπιν, τα jobs που χωρούν στην διαθέσιμη μνήμη μπαίνουν στη μνήμη και εκτελούνται. Η διαδικασία επαναλαμβάνεται όταν η εκτέλεση όλων των jobs που είναι στην μνήμη τερματιστεί. Η μέθοδος αυτή δεν κάνει καλή διαχείριση της μνήμης γιατί η μνήμη που απελευθερώνεται από κάποιο job πού τερματίστηκε δεν μπορεί να ξαναχρησιμοποιηθεί πριν τερματισθούν όλα τα jobs που βρίσκονται στη μνήμη.

### 2.2.2 Dynamic allocation καί reallocation

Κατά τη δυναμική κατανομή της μνήμης, το λειτουργικό σύστημα διατηρεί έναν πίνακα με τη θέση και το μέγεθος των ελεύθερων τμημάτων της μνήμης. Όταν ένα νέο job είναι έτοιμο για εκτέλεση, τα ελεύθερα τμήματα της μνήμης ελέγχονται για να βρεθεί κάποιο που να χωράει το job (Σχήμα 2.3). Στην περίπτωση αυτή, το τμήμα αυτό απονέμεται στο job το οποίο μπορεί τώρα να αρχίσει να εκτελείται. Όταν η εκτέλεση ενός job τερματισθεί, το

τμήμα της μνήμης πού καταλάμβανε σημειώνεται ως ελεύθερο και καταχωρείται στον πίνακα της ελεύθερης μνήμης.




Σχήμα 2.3 Partition allocation and deallocation

Για την απονομή κάποιου τμήματος της ελεύθερης μνήμης σε ένα νέο job χρησιμοποιείται ένας από τους εξείς δύο αλγόριθμους:

- **First fit algorithm**. Το πρώτο τμήμα της ελεύθερης μνήμης που μπορεί να χωρέσει το job απονέμεται στο job αυτό.
- **Best fit algorithm**. Το μικρότερο τμήμα της ελεύθερης μνήμης πού μπορεί να χωρέσει το job απονέμεται στο job αυτό.

Είναι προφανές ότι ο δεύτερος αλγόριθμος κάνει καλύτερη διαχείριση της μνήμης από τον πρώτο γιατί δημιουργεί μικρότερα αχρησιμοποίητα τμήματα μνήμης. Και με τους δύο αλγόριθμους όμως, μετά από κάποιο διάστημα λειτουργίας δημιουργείται ο λεγόμενος κατακερματισμός (fragmentation) της μνήμης. Υπάρχουν, δηλαδή, πολλά μικρά τμήματα ελεύθερης μνήμης πού δε μπορούν να χρησιμοποιηθούν από κανένα job γιατί είναι πολύ μικρά. Το λειτουργικό σύστημα εκτελεί τότε μια διαδικασία σύμπτυξης (compaction) της μνήμης μεταφέροντας τα jobs που υπάρχουν στην μνήμη το ένα μετά το άλλο. Αυτό δημιουργεί ένα μόνο μεγάλο τμήμα ελεύθερης μνήμης στο τέλος της διαθέσιμης μνήμης.

Φυσικά, για να μπορέσει να εκτελεσθεί αυτή η σύμπτυξη τα προγράμματα θα πρέπει να μπορούν να επανατοποθετηθούν σε άλλη θέση της μνήμης. Αυτό γίνεται όταν τα προγράμματα είναι relocatable, αναφέρονται, δηλαδή, σε σχετικές θέσεις της μνήμης και όχι σε απόλυτες.

## 2.3 Ιδεατή μνήμη και κρυφή μνήμη (virtual and cashe memory)

Η ιδεατή μνήμη είναι μια από τις πιο σημαντικές λειτουργίες ενός λειτουργικού συστήματος που κάνει διαχείριση μνήμης. Η βασική ιδέα είναι να δώσει τη δυνατότητα σε ένα πρόγραμμα να χρησιμοποιήσει περισσότερες θέσεις μνήμης από αυτές πού είναι διαθέσιμες στη φυσική μνήμη. Μια διεύθυνση που παράγεται από ένα πρόγραμμα του χρήστη λέγεται ιδεατή διεύθυνση. Το σύνολο όλων των ιδεατών διευθύνσεων αποτελεί τον ιδεατό χώρο μνήμης. Η πραγματικά υπάρχουσα (ή φυσική) μνήμη έχει αντίστοιχα ένα συγκεκριμένο σύνολο διευθύνσεων πού αποτελούν το φυσικό χώρο μνήμης. Σε ένα σύστημα που χρησιμοποιεί ιδεατή μνήμη, ο ιδεατός χώρος μνήμης είναι πολύ μεγαλύτερος από το φυσικό χώρο.

Η ιδεατή μνήμη απελευθερώνει τον προγραμματιστή από τους περιορισμούς της φυσικής μνήμης, έτσι ώστε να μπορεί να γράψει προγράμματα που να χρησιμοποιούν τεράστιο χώρο. Θα πρέπει να υπάρχει όμως κάποιος μηχανισμός πού να χωρίζει τον μεγάλο ιδεατό χώρο μνήμης ενός προγράμματος σε μικρότερα τμήματα που να χωρούν στη φυσική μνήμη. Αυτό μπορεί να γίνει είτε από τον ίδιο τον προγραμματιστή είτε από το λειτουργικό σύστημα. Συνήθως είναι το λειτουργικό σύστημα πού αναλαμβάνει την διαχείριση της ιδεατής μνήμης έτσι ώστε ο προγραμματιστής να μην χρειάζεται να γνωρίζει τις ιδιαιτερότητες διαχείρισης της μνήμης. Υπάρχουν δύο τρόποι διαχείρισης της ιδεατής μνήμης:

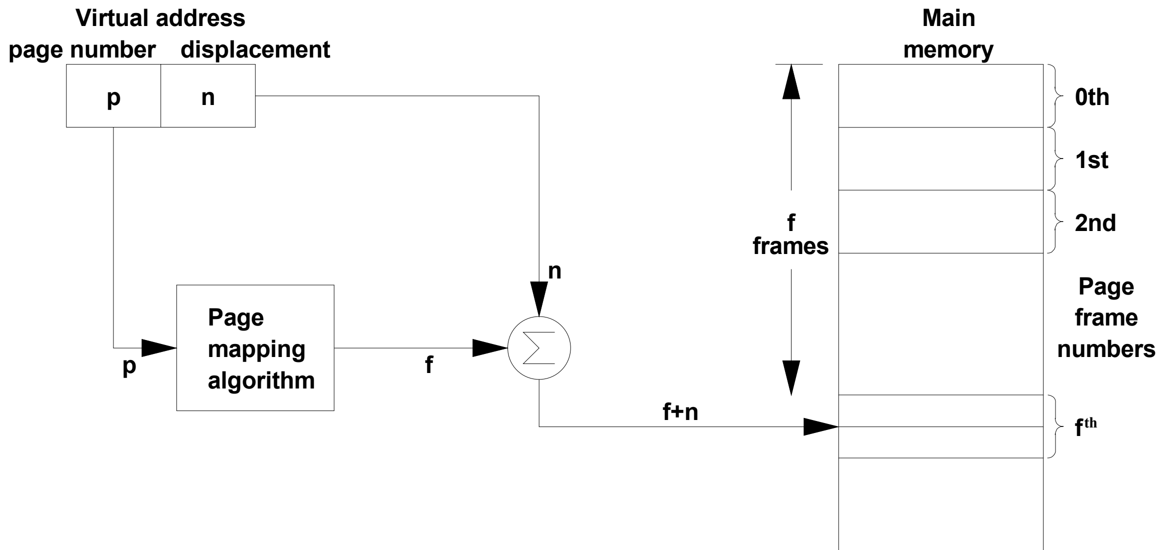
- Συστήματα σελίδων (paging systems)
- Συστήματα τμημάτων (Segmentation systems)

### 2.3.1 Paging systems

Σε ένα σύστημα σελίδων, ο ιδεατός χώρος μνήμης χωρίζεται σε blocks ίσου μεγέθους που λέγονται σελίδες. Η φυσική μνήμη χωρίζεται αντίστοιχα σε blocks ίσου μεγέθους που λέγονται πλαίσια. Το μέγεθος της σελίδας είναι ίσο με το μέγεθος του πλαισίου.

Στα συστήματα σελίδων, η ιδεατή διεύθυνση μπορεί να θεωρηθεί σαν ένα διατεταγμένο ζεύγος $[p,n]$, όπου $p$ είναι ο αριθμός σελίδας και $n$ είναι ο αριθμός της λέξης μέσα στη σελίδα. Το $n$ λέγεται displacement ή offset. Μία σελίδα $p$ του προγράμματος ενός χρήστη μπορεί να τοποθετηθεί σε οποιοδήποτε πλαίσιο $f$ της φυσικής μνήμης. Μιας και ολόκληρος ο ιδεατός χώρος του προγράμματος μπορεί να μην χωράει στην φυσική μνήμη, ένα αντίγραφο του ιδεατού χώρου διατηρείται πάντα σε δευτερεύουσα μνήμη. Το πρόγραμμα έχει προσπέλαση σε μία σελίδα, αν η σελίδα αυτή βρίσκεται στη κυρίως μνήμη. Σε ένα σύστημα σελίδων, οι σελίδες έρχονται από την δευτερεύουσα μνήμη στη κυρίως μνήμη δυναμικά.

Όλες οι ιδεατές διευθύνσεις μνήμης του προγράμματος ενός χρήστη, πρέπει να μετατραπούν σε φυσικές διευθύνσεις της κυρίως μνήμης όπως φαίνεται στο σχήμα 2.4. Όταν το πρόγραμμα που τρέχει κάνει αναφορά στη διεύθυνση $q=[p,n]$ της ιδεατής μνήμης, ο αλγόριθμος μετάφρασης βρίσκει ότι η σελίδα $p$ βρίσκεται στο φυσικό πλαίσιο $f$. Η φυσική διεύθυνση καθορίζεται από το συνδυασμό των $f$ και $n$.




Σχήμα 2.4 Μετάφραση ιδεατής διεύθυνσης σε φυσική διεύθυνση.

Η μετάφραση αυτή μπορεί να γίνει μόνο όταν η απαιτούμενη σελίδα βρίσκεται σε κάποιο πλαίσιο της φυσικής μνήμης. Στην πράξη όμως, αυτό δεν συμβαίνει πάντα. Όταν η απαιτούμενη σελίδα δεν βρίσκεται στη κυρίως μνήμη, έχουμε σφάλμα σελίδας (page fault). Αυτό προκαλεί στην πραγματικότητα ένα interrupt για το λειτουργικό σύστημα. Η υπορρουτίνα που χειρίζεται αυτό το interrupt λέγεται page-fault handler, και τα βήματα πού ακολουθεί είναι τα εξής:

1. Η θέση της απαιτούμενης σελίδας στη δευτερεύουσα μνήμη διαβάζεται από τον πίνακα σελίδων.
2. Η σελίδα *p* μεταφέρεται από τη δευτερεύουσα μνήμη σε ένα πλαίσιο της κυρίως μνήμης.
3. Ο πίνακας σελίδων ενημερώνεται με τον αριθμό του πλαισίου που περιέχει την σελίδα *p*.

Όταν η υπορρουτίνα μεταφοράς των σελίδων τερματισθεί, ο έλεγχος μεταφέρεται στο πρόγραμμα του χρήστη. Υπάρχουν διάφορες πολιτικές για τη μεταφορά σελίδων από τη δευτερεύουσα μνήμη στην κυρίως μνήμη. Η πολιτική πού καθορίζει ποιά σελίδα θα μεταφερθεί από την δευτερεύουσα μνήμη, λέγεται fetch policy. Εδώ θα πρέπει να σημειωθεί ότι οι σελίδες δεν μεταφέρονται μόνο όταν ζητηθούν. Ανάλογα με τον αλγόριθμο του fetch policy μπορεί να έχουμε μια μεταφορά σελίδας από τη δευτερεύουσα μνήμη στην κυρίως μνήμη, ακόμα και αν δεν έχει ζητηθεί.

Αν όλα τα διαθέσιμα πλαίσια της κυρίως μνήμης είναι κατειλημμένα όταν μια σελίδα πρέπει να μεταφερθεί από τη δευτερεύουσα μνήμη, τότε κάποιο από τα πλαίσια πρέπει να αφαιρεθεί από την κυρίως μνήμη και να τοποθετηθεί πίσω στην δευτερεύουσα. Η πολιτική πού καθορίζει ποιό πλαίσιο θα αφαιρεθεί από την κυρίως μνήμη λέγεται replacement policy.

### 2.3.2 Segmentation systems

Η τεχνική του paging που είδαμε μέχρι τώρα είναι μια μονοδιάστατη τεχνική γιατί η διεύθυνση της ιδεατής μνήμης αυξάνει γραμμικά από το 0 μέχρι κάποια τιμή *M*. Σε μερικές περιπτώσεις, είναι σκόπιμο να έχουμε μια πολυδιάστατη τεχνική. Αυτό ακριβώς κάνει ένα segmentation system.

Στα segmentation systems κάθε λογική ενότητα, όπως είναι το stack, μια υπορρουτίνα, ή ένας πίνακας δεδομένων, μπορεί να έχει το δικό της χώρο ιδεατής μνήμης. Κάθε χώρος ιδεατής μνήμης λέγεται segment. Αφού και κάθε segment αναφέρεται σε διαφορετικό χώρο ιδεατής μνήμης, μπορεί να αυξηθεί ή να μειωθεί ανεξάρτητα από τα άλλα segments. Τα στοιχεία κάθε segment ενός συστήματος διατηρούνται σε έναν πίνακα που λέγεται segmentation table. Κάθε εγγραφή στον πίνακα αυτό περιλαμβάνει τα ακόλουθα στοιχεία:

1. Segment base address $b$ (η αρχική διεύθυνση του segment στην κυρίως μνήμη).
2. Μήκος του segment $l$
3. Segment presence bit (καθορίζει αν το segment βρίσκεται στην κυρίως μνήμη ή όχι)
4. Protection bits (καθορίζουν προστασία για το segment).

Είναι προφανές ότι μπορούμε να έχουμε segments με διαφορετικό μήκος το καθένα. Αν το μήκος $l$ από όλα τα segments είναι το ίδιο, τότε το σύστημα είναι παρόμοιο με το paging σύστημα.

Η μετάφραση της ιδεατής διεύθυνσης γίνεται με παρόμοιο τρόπο με αυτόν του paging. Στην περίπτωση των segments όμως, ένας επιπλέον έλεγχος πρέπει να γίνει για την περίπτωση που το ofsset ή displacement είναι μεγαλύτερο από το μήκος του segment. Αυτό θα προκαλέσει πάλι ένα interrupt για το λειτουργικό σύστημα. Συνοπτικά, τα interrupts που μπορεί να έχουμε σε ένα segmentation system είναι:

1. Segment fault. Το interrupt αυτό ενεργοποιείται όταν το απαιτούμενο segment δεν βρίσκεται στην κυρίως μνήμη.
2. Address violation trap. Το interrupt αυτό ενεργοποιείται όταν η απαιτούμενη διεύθυνση μέσα στο segment είναι μεγαλύτερη από το μήκος του segment.
3. Protection violation trap. Το interrupt αυτό ενεργοποιείται όταν υπάρξει παραβίαση της προστασίας του segment.

Τα πιο συνηθισμένα πρωτόκολλα προστασίας ενός segment είναι:

1. Read only (το πρόγραμμα μπορεί μόνο να διαβάσει από αυτό το segment).
2. Execute only (το πρόγραμμα μπορεί μόνο να εκτελέσει εντολές από αυτό το segment).
3. Read and execute only (το πρόγραμμα μπορεί μόνο να διαβάσει ή να εκτελέσει εντολές από αυτό το segment).
4. Unlimited access (απεριόριστη προσπέλαση).
5. No access (καμία προσπέλαση).

Σε μερικά μεγάλα συστήματα χρησιμοποιούνται και οι δύο τεχνικές ιδεατής μνήμης. Πρώτα καθορίζονται τα segments ανάλογα με τις λογικές ενότητες του προγράμματος, και κατόπιν τα segments χωρίζονται σε σελίδες. Χρησιμοποιώντας την δυναμική τοποθέτηση των σελίδων και των segments στην κυρίως μνήμη, η εξής διαδικασία ακολουθείται όταν ζητηθεί μια διεύθυνση:

Εξετάζεται πρώτα το segment table για να διαπιστωθεί αν το απαιτούμενο segment βρίσκεται στην κυρίως μνήμη. Αν ναι, τότε εξετάζεται ο πίνακας σελίδων για να διαπιστωθεί αν η απαιτούμενη σελίδα βρίσκεται σε κάποιο πλαίσιο της μνήμης. Αν η σελίδα βρίσκεται στη μνήμη τότε γίνεται η μετάφραση της διεύθυνσης. Αν η απαιτούμενη σελίδα δεν βρίσκεται στην κυρίως μνήμη, τότε μεταφέρεται από τη δευτερεύουσα μνήμη και ενημερώνεται ο πίνακας σελίδων. Αν τέλος το απαιτούμενο segment δεν βρίσκεται στην κυρίως μνήμη, τότε

δημιουργείται χώρος γι' αυτό και η απαιτούμενη σελίδα του μεταφέρεται από τη δευτερεύουσα μνήμη στην κυρίως μνήμη.

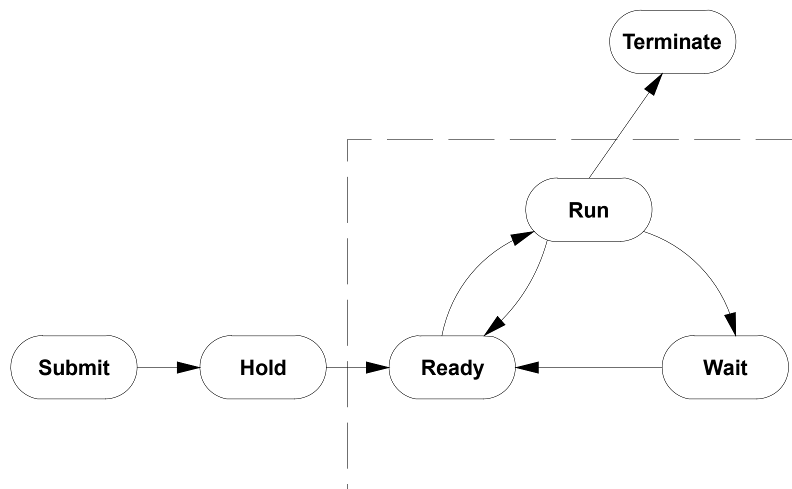
# 3 Λειτουργίες διαχείρισης του processor

Οι λειτουργίες διαχείρισης του processor καθορίζουν ποιό πρόγραμμα θα εκτελεσθεί, πότε θα αρχίσει η εκτέλεσή του, για πόσο χρόνο θα απασχολήσει τον processor κλπ. Γενικά, οι λειτουργίες αυτές μπορούν να χωρισθούν σε δύο κατηγορίες:

1. Job scheduling είναι η λειτουργία πού ελέγχει πότε το πρόγραμμα θα τοποθετηθεί στην κυρίως μνήμη και θα εκτελεσθεί. Αν το σύστημα εφαρμόζει επίσης time-sharing τότε ο job scheduler καθορίζει επίσης πότε το πρόγραμμα θα βγει για λίγο από την κυρίως μνήμη έτσι ώστε και κάποιο άλλο πρόγραμμα να έχει την ευκαιρία να εκτελεσθεί για κάποιο χρονικό διάστημα.
2. Multiprogramming είναι η λειτουργία πού επιτρέπει σε πολλά προγράμματα να βρίσκονται ταυτόχρονα στη μνήμη και να εκτελούνται εναλλάξ μέχρι να τερματισθούν. Η κύρια λειτουργία του λειτουργικού συστήματος που κάνει multiprogramming λέγεται Process scheduling.

## 3.1 Job scheduling

Από την στιγμή πού ένα πρόγραμμα θα δοθεί στο υπολογιστικό σύστημα μέχρι τη στιγμή πού η εκτέλεσή του θα τελειώσει και τα αποτελέσματα θα δοθούν στον χρήστη, περνάει από διάφορα στάδια επεξεργασίας. Τα στάδια αυτά φαίνονται διαγραμματικά στο Σχήμα 3.1.




Σχήμα 3.1 Στάδια επεξεργασίας ενός προγράμματος.

Το μέρος του λειτουργικού συστήματος που κάνει το job scheduling (ο job scheduler δηλαδή) ασχολείται μόνο με τη μετάβαση από το στάδιο hold στο στάδιο ready ή και το αντί-στροφο αν το σύστημα εφαρμόζει time-sharing.

Πιο αναλυτικά, τα διάφορα στάδια ενός προγράμματος είναι:

1. **Submit**. Το πρόγραμμα εισέρχεται στο σύστημα για εκτέλεση.
2. **Hold**. Το πρόγραμμα περιμένει στην δευτερεύουσα μνήμη μέχρι να τελειώσει κάποιο πρόγραμμα πού εκτελείται.
3. **Ready**. Το πρόγραμμα τοποθετείται στην κυρίως μνήμη και περιμένει την σειρά του να εκτελεσθεί.

4. **Run**. Το πρόγραμμα εκτελείται.
5. **Wait**. Το πρόγραμμα περιμένει να τελειώσει κάποια διαδικασία εισόδου-εξόδου.
6. **Terminate**. Η εκτέλεση του προγράμματος έχει τελειώσει.

Οι βασικοί σκοποί του job scheduling είναι:

α) Να εκτελεσθούν όσο το δυνατόν περισσότερα προγράμματα κατά την διάρκεια της ημέρας.

β) Να ελαχιστοποιηθεί ο χρόνος ανάμεσα στην παράδοση ενός προγράμματος για εκτέλεση και την επιστροφή των αποτελεσμάτων στον χρήστη.

γ) Να διατηρείται ο processor απασχολημένος όσο το δυνατόν γίνεται.

Μερικές φορές οι σκοποί αυτοί είναι αντικρουόμενοι. Για παράδειγμα, το α) ευνοεί τα μικρά προγράμματα, ενώ το γ) ευνοεί τα μεγάλα προγράμματα που εκτελούν πολύπλοκους υπολογισμούς. Για τον λόγο αυτό μπορούν να εφαρμοσθούν διάφορες πολιτικές από τον job scheduler. Σε περιβάλλοντα που δεν κάνουν multiprogramming οι δυο πιο απλές πολιτικές που μπορούν να χρησιμοποιηθούν είναι: 1) FIFO (First In First Out), δηλαδή το πρώτο πρόγραμμα πού θα μπει στο σύστημα θα είναι και το πρώτο που θα εκτελεσθεί, και 2) Το μικρότερο πρόγραμμα πρώτα που σημαίνει ότι αν δύο προγράμματα περιμένουν για εκτέλεση, το πιο μικρό από τα δύο θα εκτελεσθεί πρώτο.

Η απόδοση κάθε πολιτικής ή αλγόριθμου για job scheduling εκτιμάται με βάση το χρόνο επιστροφής (turnaround time), τον πραγματικό χρόνο δηλαδή που περνάει από την στιγμή πού θα μπει ένα πρόγραμμα στο σύστημα μέχρι τη στιγμή που θα βγει από αυτό. Ο χρόνος αυτός είναι συνήθως πολύ μεγαλύτερος από τον πραγματικό χρόνο εκτέλεσης (run time) του προγράμματος. Επειδή είναι φυσικό ένα μεγάλο πρόγραμμα να τελειώσει σε μεγαλύτερο χρόνο από ένα μικρό, τις περισσότερες φορές σαν μέτρο σύγκρισης χρησιμοποιείται ο ανηγμένος χρόνος επιστροφής (weighted turnaround time) πού είναι ο χρόνος επιστροφής ενός προγράμματος διαιρούμενος με τον χρόνο εκτέλεσης του προγράμματος.

Μερικές από τις πολιτικές δίνουν καλύτερο μέσο χρόνο επιστροφής, ενώ άλλες δίνουν καλύτερο μέσο ανηγμένο χρόνο επιστροφής. Ο καλύτερος δυνατός μέσος ανηγμένος χρόνος επιστροφής επιτυγχάνεται όταν εκτός από κάποιο job scheduling, το σύστημα εφαρμόζει επίσης multiprogramming.

## 3.2 Process scheduling

Το process scheduling ελέγχει και καθορίζει τις μεταβάσεις ενός προγράμματος από τα στάδια Ready, Run και Wait. Η μετάβαση από το στάδιο run στο στάδιο wait γίνεται όταν ένα πρόγραμμα απαιτήσει κάποια διαδικασία εισόδου-εξόδου. Μόλις αυτή η διαδικασία περατωθεί, το πρόγραμμα περνάει στο στάδιο ready. Δεν είναι όμως μόνο αυτή η περίπτωση κατά την οποία ένα πρόγραμμα θα μεταβεί στο στάδιο ready. Σε ένα περιβάλλον multiprogramming, ο processor scheduler φροντίζει ώστε όλα τα προγράμματα που βρίσκονται στο στάδιο ready να έχουν την ευκαιρία να εκτελεσθούν για λίγο χρονικό διάστημα.

Ο processor scheduler καθορίζει ένα κάποιο μικρό χρονικό διάστημα (συνήθως 100 ms) κατά το οποίο το πρόγραμμα έχει στην πλήρη κατοχή του το CPU. Αν το διάστημα αυτό εξαντληθεί και το πρόγραμμα δεν έχει τερματισθεί, ούτε έχει ζητήσει κάποιο I/O, τότε επιστρέφει στο στάδιο ready και κάποιο άλλο πρόγραμμα από αυτά που ήταν στο στάδιο ready συνεχίζει την εκτέλεσή του. Το χρονικό αυτό διάστημα λέγεται time quantum.

Υπάρχουν διάφορες πολιτικές για την απονομή των quantums στα διάφορα προγράμματα. Οι πιο συνηθισμένες απ' αυτές είναι:

1. **Round robin**. Κάθε πρόγραμμα με τη σειρά του παίρνει από ένα time quantum.
2. **Inverse of remainder of quantum**. Αν το πρόγραμμα χρησιμοποίησε όλο το quantum την προηγούμενη φορά, πηγαίνει στο τέλος της ουράς. Αν χρησιμοποίησε το μισό (γιατί περίμενε γιά I/O), πηγαίνει στο μέσο της ουράς.
3. **Multiple-level feedback round robin**. Όταν ένα νέο πρόγραμμα μπαίνει στο σύστημα, εκτελείται συνεχώς για όσα time quantums έχουν χρησιμοποιήσει όλα τα άλλα προγράμματα στο σύστημα. Μετά εκτελείται με κανονικό round robin.
4. **Priority**. Το πρόγραμμα με την μεγαλύτερη προτεραιότητα επιλέγεται να εκτελεσθεί.
5. **Limited round robin**. Τα προγράμματα εκτελούνται με τη διαδικασία του round robin μέχρι να χρησιμοποιήσουν κάποιο προκαθορισμένο αριθμό quantums. Μετά εκτελούνται μόνο αν δεν υπάρχει άλλο πρόγραμμα στο σύστημα.
6. **System balance**. Δίνεται προτεραιότητα στα προγράμματα πού απαιτούν πολύ I/O.
7. **Preferred interactive**. Δίνεται προτεραιότητα στα προγράμματα πού εκτελούνται από τερματικά.
8. **Job merits**. Αυτή είναι η πιο πολύπλοκη τεχνική απονομής χρόνου στα προγράμματα που βρίσκονται στο στάδιο ready. Το λειτουργικό σύστημα απονέμει δυναμικά προτεραιότητες στα προγράμματα εξετάζοντας τις απαιτήσεις τους γιά I/O, τον χρόνο πού έχουν παραμείνει στο σύστημα, τον αριθμό των περιφερειακών που χρησιμοποιούν κλπ.

## 3.3 Deadlocks

Κατά τη διάρκεια εκτέλεσης ενός προγράμματος, αυτό κάνει χρήση διάφορων περιφερειακών. Όταν το πρόγραμμα χρειασθεί κάποιο περιφερειακό, το λειτουργικό σύστημα ελέγχει αν το περιφερειακό είναι διαθέσιμο. Στην περίπτωση που το περιφερειακό δεν είναι διαθέσιμο το πρόγραμμα θα πρέπει να περιμένει. Αυτό λέγεται μπλοκάρισμα (block) του προγράμματος. Σε ορισμένες περιπτώσεις, είναι δυνατόν το πρόγραμμα A να έχει στην διάθεσή του κάποιο περιφερειακό και να απαιτεί ένα περιφερειακό που κατέχεται από το πρόγραμμα B. Αν τώρα το πρόγραμμα B απαιτήσει το περιφερειακό που κατέχει το A έχουμε μια κατάσταση που λέγεται deadlock (ή deadly embrace). Κανένα από τα δύο προγράμματα δεν μπορεί να βγει από την κατάσταση blocked γιατί το περιφερειακό πού απαιτεί δεν θα απελευθερωθεί ποτέ από το άλλο πρόγραμμα.

Το deadlock είναι μια ανεπιθύμητη κατάσταση και γι' αυτό αναπτύχθηκαν διάφοροι αλγόριθμοι για την επίλυσή του. Ο πιο απλός αλγόριθμος απονέμει στα προγράμματα πού πρόκειται να εκτελεσθούν, όλα τα περιφερειακά πού θα ζητήσουν κατά την εκτέλεσή τους. Η μέθοδος αυτή απαιτεί γνώση των μελλοντικών απαιτήσεων κάθε προγράμματος και επιπλέον, γίνεται κακή χρήση των περιφερειακών αφού κατακρατούνται καθ' όλη την διάρκεια εκτέλεσης του προγράμματος ακόμα και αν αυτό τα χρειασθεί μόνο για ελάχιστο χρόνο.

Ένας άλλος αλγόριθμος χρησιμοποιεί μια διάταξη των περιφερειακών αριθμώντας τα από 1 μέχρι το μέγιστό τους αριθμό (π.χ. 1=printer, 2=plotter, 3=disk κλπ.). Ένα πρόγραμμα μπορεί να απαιτήσει περιφερειακά μόνο με αύξουσα σειρά. Για παράδειγμα, αν ένα πρόγραμμα κατέχει ήδη το περιφερειακό 7 δεν μπορεί να απαιτήσει το περιφερειακό 4, γιατί το 4 είναι μικρότερο απ' το 7. Το σύστημα αυτό έχει πολλά προβλήματα. Αν για παράδειγμα το tape

είναι άμεσα απαραίτητο και ο printer δεν χρειάζεται παρά πολύ αργότερα, το πρόγραμμα θα πρέπει πρώτα να ζητήσει τον printer και μετά το tape.

Η καλύτερη λύση του προβλήματος του deadlock έχει αποδειχθεί ότι είναι το detect and recover. Σύμφωνα με την τεχνική αυτή, το deadlock επιτρέπεται να εμφανισθεί, αλλά το λειτουργικό σύστημα έχει τη δυνατότητα να το λύσει. Σε τακτά χρονικά διαστήματα, το λειτουργικό σύστημα ελέγχει την αλληλοσυσχέτιση των προγραμμάτων για deadlock και αν διαπιστωθεί ότι υπάρχει deadlock, το ένα από τα δύο προγράμματα υποχρεώνεται να αποδώσει τα περιφερειακά πού κατέχει, έτσι ώστε το άλλο πρόγραμμα να μπορέσει να συνεχίσει.

# 4 Αλγόριθμοι (πολιτικές) χρονοδρομολόγησης

Οι χρονοδρομολογητές ενός λειτουργικού συστήματος, χρησιμοποιούν αλγόριθμους, οι οποίοι προσπαθούν να βελτιστοποιήσουν διάφορα χαρακτηριστικά του συστήματος. Στα χαρακτηριστικά αυτά περιλαμβάνονται ο βαθμός χρήσης (Utilization) της CPU, ο χρόνος επιστροφής (turnaround time), ο χρόνος απόκρισης (response time), κλπ. Στο παράδειγμα που ακολουθεί γίνεται μια προσπάθεια σύγκρισης μερικών αλγορίθμων κατά την εκτέλεση του ιδίου σετ διεργασιών.

Ορίζουμε τις παρακάτω μεταβλητές:

**Χρόνος Εκτέλεσης (Running Time ή RT):** Ο πραγματικός χρόνος που απαιτείται για την εκτέλεση μιάς διεργασίας. Είναι ο χρόνος που χρειάζεται για να "τρέξει" μια διεργασία αν ήταν η μόνη διεργασία του συστήματος (είχε δηλαδή όλη τη CPU δική της).

**Χρόνος Επιστροφής (Turnaround Time ή TT):** Ο χρόνος που μεσολαβεί από την παράδοση της εργασίας προς εκτέλεση μέχρι την παραλαβή των αποτελεσμάτων.

**Ανηγμένος Χρόνος επιστροφής (Weighted Turnaround Time ή WTT):** Ο λόγος του χρόνου επιστροφής μιάς διεργασίας προς τον χρόνο εκτέλεσης της διεργασίας

**Χρόνος Αναμονής (Waiting Time ή WT):** Είναι ο χρόνος για τον οποίο περιμένει μια διεργασία χωρίς να εκτελείται. Ισούται με το χρόνο επιστροφής μείον τον χρόνο εκτέλεσης.

**Ανηγμένος Χρόνος Αναμονής (Weighted Waiting Time ή WWT):** Ο λόγος του χρόνου αναμονής προς το χρόνο εκτέλεσης κάθε διεργασίας.

Ορίζονται επίσης και οι μέσοι χρόνοι ως η μέση τιμή των παραπάνω χρόνων.

Για το παράδειγμα που εξετάζουμε, θα υπολογισθούν οι χρόνοι αυτοί για το ίδιο σετ διεργασιών και για τρεις διαφορετικούς αλγόριθμους χρονοδρομολόγησης.

Έστω ένα σύστημα που είναι αδρανές (idle) στον χρόνο 0, έρχονται τρεις εργασίες για εκτέλεση με τα παρακάτω χαρακτηριστικά:

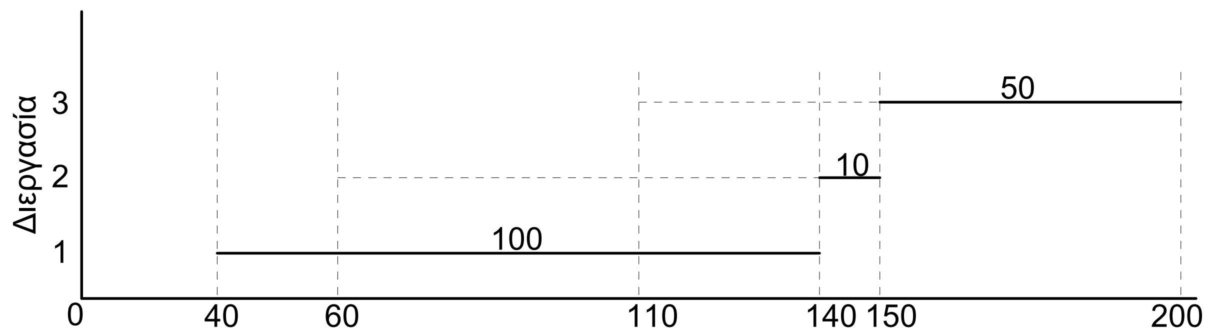
| Job | Χρόνος άφιξης | Χρόνος εκτέλεσης |
|---|---|---|
| 1 | 40 sec | 100 sec |
| 2 | 60 sec | 10 sec |
| 3 | 110 sec | 50 sec |

Για τον υπολογισμό των χρόνων είναι πολύ σημαντικό να σχεδιασθούν σωστά τα σχεδιαγράμματα χρονισμού από τα οποία εξάγονται αμέσως οι απαιτούμενοι χρόνοι.

### α) Αλγόριθμος FCFS (First Come First Served)

Σύμφωνα με τον αλγόριθμο αυτό, ο οποίος είναι αλγόριθμος *μη προεκχώρησης*, η διεργασία που έρχεται πρώτη, θα εκτελεσθεί πρώτη χωρίς να μπορεί να διακοπεί από άλλη διεργασία. Το χρονοδιάγραμμα των διεργασιών δίνεται στο σχήμα 4.1.

Στο διάγραμμα αυτό, για κάθε διεργασία σημειώνεται με διακεκομμένη γραμμή ο χρόνος αναμονής και με συνεχή γραμμή ο χρόνος εκτέλεσης. Ο χρόνος εκτέλεσης δίνεται αριθμητικά και πάνω στη συνεχή γραμμή εκτέλεσης.




Σχήμα 4.1 Χρονοδιάγραμμα αλγόριθμου FCFS.

Οι χρόνοι υπολογίζονται ως εξής:

$TT_1$=140-40=100 $\quad$ $WT_1=TT_1-RT_1$=100-100=0

$TT_2$=150-60=90 $\quad$ $WT_2=TT_2-RT_2$=90-10=80

$TT_3$=200-110=90 $\quad$ $WT_3=TT_3-RT_3$=90-50=40

Οι μέσοι χρόνοι είναι:

$$ATT = \frac{TT_1 + TT_2 + TT_3}{3} = \frac{100+90+90}{3} = 93.33$$

και

$$AWT = \frac{WT_1 + WT_2 + WT_3}{3} = \frac{0+80+40}{3} = 40$$

Οι ανηγμένοι χρόνοι είναι:

$$WTT_1 = \frac{100}{100} = 1 \qquad WWT_1 = \frac{0}{100} = 0$$

$$WTT_2 = \frac{90}{10} = 9 \qquad WWT_2 = \frac{80}{10} = 8$$

$$WTT_3 = \frac{90}{50} = 1.8 \qquad WWT_3 = \frac{40}{50} = 0.8$$

Τέλος, οι μέσοι ανηγμένοι χρόνοι είναι:

$$AWTT = \frac{WTT_1 + WTT_2 + WTT_3}{3} = \frac{1+9+1.8}{3} = 3.93$$

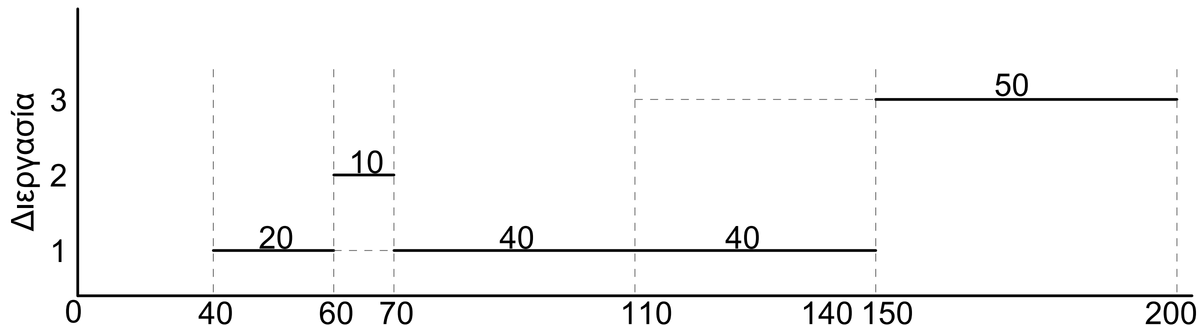
και

$$AWWT = \frac{WWT_1 + WWT_2 + WWT_3}{3} = \frac{0+8+0.8}{3} = 2.93$$

Παρατηρούμε ότι ο μέσος ανηγμένος χρόνος αναμονής (AWWT) είναι ίσος με τον μέσο ανηγμένο χρόνο επιστροφής (AWTT) μείον 1, όπως μπορεί εύκολα να αποδειχθεί μαθηματικά.

**α) Αλγόριθμος SRTN (Shortest Remaining Time Next)**

Σύμφωνα με τον αλγόριθμο αυτό, ο οποίος είναι αλγόριθμος *προεκχώρησης*, η διεργασία που έχει τον συντομότερο εναπομείναντα χρόνο εκτέλεσης θα εκτελεσθεί πρώτα, διακόπτοντας τυχόν εκτελούμενη εργασία με μεγαλύτερο εναπομείναντα χρόνο εκτέλεσης. Το χρονοδιάγραμμα των διεργασιών δίνεται στο σχήμα 4.2.




Σχήμα 4.2 Χρονοδιάγραμμα αλγόριθμου SRTN.

Οι χρόνοι υπολογίζονται ως εξής:

$TT_1$=150-40=110 $WT_1$=$TT_1$-$RT_1$=110-100=10

$TT_2$=70-60=10 $WT_2$=$TT_2$-$RT_2$=10-10=0

$TT_3$=200-110=90 $WT_3$=$TT_3$-$RT_3$=90-50=40

Οι μέσοι χρόνοι είναι:

$$ATT = \frac{TT_1 + TT_2 + TT_3}{3} = \frac{110 + 10 + 90}{3} = 70$$

και

$$AWT = \frac{WT_1 + WT_2 + WT_3}{3} = \frac{10 + 0 + 40}{3} = 16.67$$

Οι ανηγμένοι χρόνοι είναι:

$$WTT_1 = \frac{110}{100} = 1.1 \qquad WWT_1 = \frac{10}{100} = 0.1$$

$$WTT_2 = \frac{10}{10} = 1 \qquad WWT_2 = \frac{0}{10} = 0$$

$$WTT_3 = \frac{90}{50} = 1.8 \qquad WWT_3 = \frac{40}{50} = 0.8$$

Τέλος, οι μέσοι ανηγμένοι χρόνοι είναι:

$$AWTT = \frac{1.1 + 1 + 1.8}{3} = 1.3$$

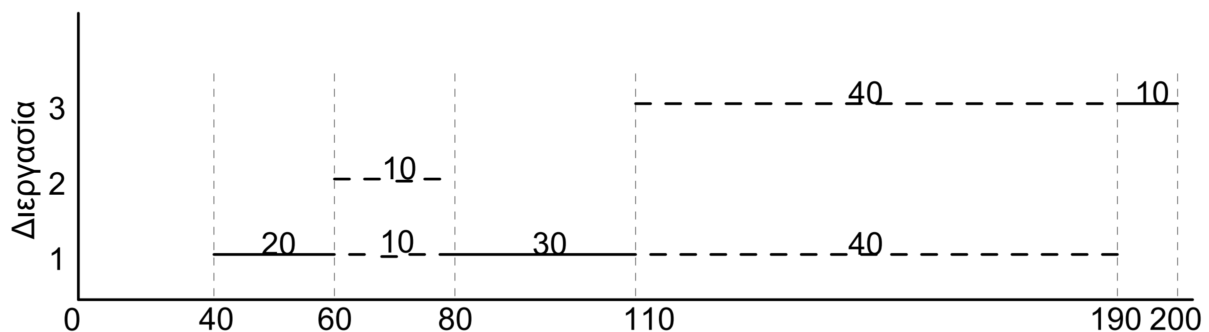
και

$$AWWT = \frac{0.1 + 0 + 0.8}{3} = 0.3$$

Παρατηρούμε ότι με τον αλγόριθμο αυτό έχουν βελτιωθεί αισθητά οι μέσοι ανηγμένοι χρόνοι., σε σχέση με τον αλγόριθμο FCFS. Πρέπει να σημειωθεί όμως, ότι για να δουλέψει ο αλγόριθμος αυτός, θα πρέπει ο χρονοδρομολογητής να γνωρίζει το χρόνο εκτέλεσης κάθε διεργασίας, πράγμα που δεν συμβαίνει γενικά.

**α) Αλγόριθμος εκ περιτροπής (Round Robin)**

Σύμφωνα με τον αλγόριθμο αυτό, ο οποίος είναι αλγόριθμος *προεκχώρησης*, όσες διεργασίες βρίσκονται στο σύστημα εκτελούνται εκ περιτροπής, για ένα κβάντο χρόνου η κάθε μια. Το χρονοδιάγραμμα των διεργασιών δίνεται στο σχήμα 4.3. Στην κατασκευή του χρονοδιαγράμματος αυτού χρειάζεται προσοχή στο να σημειώνουμε το μισό χρόνο εκτέλεσης για κάθε διεργασία αν εκτελούνται δύο ταυτόχρονα, το ένα τρίτο του χρόνου αν εκτελούνται τρεις ταυτόχρονα, κ.ο.κ.




Σχήμα 4.3 Χρονοδιάγραμμα αλγόριθμου εκ περιτροπής.

Οι χρόνοι υπολογίζονται ως εξής:

$TT_1$=190-40=150 $WT_1$=$TT_1$-$RT_1$=150-100=50

$TT_2$=80-60=20 $WT_2$=$TT_2$-$RT_2$=20-10=10

$TT_3$=200-110=90 $WT_3$=$TT_3$-$RT_3$=90-50=40

Οι μέσοι χρόνοι είναι:

$$ATT = \frac{150+20+90}{3} = 86.67$$

$$AWT = \frac{50+10+40}{3} = 33.33$$

Οι ανηγμένοι χρόνοι είναι:

$$WTT_1 = \frac{150}{100} = 1.5 \qquad WWT_1 = \frac{50}{100} = 0.5$$

$$WTT_2 = \frac{20}{10} = 2 \qquad WWT_2 = \frac{10}{10} = 1$$

$$WTT_3 = \frac{90}{50} = 1.8 \qquad WWT_3 = \frac{40}{50} = 0.8$$

Τέλος, οι μέσοι ανηγμένοι χρόνοι είναι:

$$AWTT = \frac{1.5+2+1.8}{3} = 1.77$$

$$AWWT = \frac{0.5+1+0.8}{3} = 0.77$$

Παρατηρούμε ότι οι μέσοι ανηγμένοι χρόνοι είναι μικρότεροι από τον αλγόριθμο FCFS αλλά μεγαλύτεροι από τον αλγόριθμο SRTN. Σε αντίθεση όμως με τον αλγόριθμο SRTN, ο αλγόριθμος εκ περιτροπής μπορεί να χρησιμοποιηθεί και όταν ο χρονοδρομολογητής δεν γνωρίζει το χρόνο εκτέλεσης κάθε διεργασίας. Επίσης, ο αλγόριθμος εκ περιτροπής είναι περισσότερο "δίκαιος" από τον αλγόριθμο SRTN, όπως μπορεί να διαπιστωθεί ελέγχοντας τους ανηγμένους χρόνους επιστροφής. Στον αλγόριθμο SRTN ο λόγος του μεγαλύτερου προς το μικρότερο ανηγμένο χρόνο επιστροφής είναι 1,8/1=1,8, ενώ στον αλγόριθμο εκ περιτροπής ο λόγος αυτός είναι 2/1,5=1,33.

# 5 Προγραμματισμός Εισόδου/Εξόδου

Ο έλεγχος της λειτουργίας των συσκευών εισόδου/εξόδου (I/O) μπορεί να γίνει με δύο τρόπους: (α) έλεγχο με πρόγραμμα, και (β) έλεγχο με διακοπές (interrupts).

Στην πρώτη περίπτωση κατασκευάζουμε ένα πρόγραμμα που εκτελεί την είσοδο ή έξοδο για κάθε χαρακτήρα διαδοχικά. Ανάμεσα στην είσοδο (ή έξοδο) κάθε χαρακτήρα, το πρόγραμμα ελέγχει τη συσκευή για να διαπιστώσει αν είναι έτοιμη να στείλει (ή να δεχθεί) τον επόμενο χαρακτήρα. Με δεδομένη την τεράστια διαφορά ταχύτητος ανάμεσα στις συσκευές I/O και τον επεξεργαστή, είναι προφανές ότι το πρόγραμμα καταναλώνει τον περισσότερο χρόνο στον έλεγχο αυτό, ενώ θα μπορούσε να εκτελεί κάποια άλλη διεργασία. Αυτό έχει ως αποτέλεσμα την μείωση στο ελάχιστο του βαθμού χρήσης (utilization) του επεξεργαστή. Ένα παράδειγμα εκτύπωσης με προγραμματισμένο I/O δίνεται στο Σχήμα 5.1.

Για να αυξήσουμε το βαθμό χρήσης του επεξεργαστή προτιμούμε τον δεύτερο τρόπο ελέγχου που κάνει χρήση των διακοπών τις οποίες μπορεί να δεχθεί κάθε επεξεργαστής. Κατασκευάζοντας τις κατάλληλες ρουτίνες ελέγχου διακοπών (interrupt service routines ή ISRs), ο επεξεργαστής δεν καταναλώνει πλέον χρόνο ελέγχοντας αν η συσκευή είναι έτοιμη να στείλει (ή να δεχθεί) τον επόμενο χαρακτήρα. Ο επεξεργαστής εκτελεί πλέον κανονικά οποιαδήποτε διεργασία και όταν η συσκευή I/O είναι έτοιμη να στείλει (ή να δεχθεί) τον επόμενο χαρακτήρα, ειδοποιεί τον επεξεργαστή με την κατάλληλη διακοπή. Στο σημείο αυτό διακόπτεται η εκτέλεση της διεργασίας, εκτελείται η ρουτίνα ελέγχου της αντίστοιχης διακοπής, η οποία και διαβάζει (ή στέλνει) τον επόμενο χαρακτήρα, και κατόπιν συνεχίζεται κανονικά η εκτέλεση της διεργασίας που διεκόπει. Το παράδειγμα του σχήματος 5.1 αλλά με έλεγχο από διακοπές δίνεται στο Σχήμα 5.2.

Στο επόμενο παράδειγμα (που φαίνεται στο σχήμα 5.3) έχουμε δύο συσκευές: μία εισόδου (keyboard) και μία εξόδου (display). Έχουμε επίσης μια διεργασία που εκτελεί κάποιους υπολογισμούς (compute). Οι ρουτίνες ελέγχου διακοπών είναι η process keyboard και η process display.

Δίνονται τρεις εκδόσεις του κυρίως προγράμματος για να γίνει κατανοητός ο τρόπος επικάλυψης των υπολογισμών (compute) με τις διακοπές που κάνουν την είσοδο και έξοδο. Στην πρώτη έκδοση, δεν υπάρχει καμία επικάλυψη ανάμεσα στις λειτουργίες γιατί περιμένουμε να τελειώσει η κάθε μία πριν αρχίσει η επόμενη. Στην δεύτερη έκδοση, περιμένουμε μεν να τελειώσει η είσοδος πριν κάνουμε οτιδήποτε άλλο, αλλά υπάρχει επικάλυψη ανάμεσα στους υπολογισμούς και την έξοδο. Τέλος, στην τρίτη έκδοση υπάρχει πλήρης επικάλυψη των τριών λειτουργιών. Στο σχήμα 5.4 δίνεται το χρονοδιάγραμμα εκτέλεσης των τριών λειτουργιών. Οι λειτουργίες εισόδου/εξόδου γίνονται με διακοπές και γι' αυτό φαίνονται με διακεκομμένες γραμμές.

```
program/module computeprint;
   const
      max = ...;                  {μέγιστο μήκος εξόδου}
      outrdy = ...;               {έξοδος έτοιμη}
   type
      text = array [1..max] of char;
   Var
      data: text;
      numberof: integer;

   procedure compute(var comdata: text; var count: integer);
      ...

   procedure print(outdata: text; outcount: integer);
      Var
         outindex : integer;
      begin
         outindex := 1;
         while outcount>0 do
            begin
               while ioport_status<>outrdy do {keeptesting};
               ioport_dataout := outdata[outindex];
               outindex := outindex + 1;
               outcount := outcount -1
            end {while}
      end; {print}

   begin {computeprint}
      initialize_hardware;
      while true do {για πάντα}
         begin
            compute(data, numberof);
            print(data, numberof)
         end {while}
   end {computeprint}
```

Σχήμα 5.1 Έξοδος ελεγχόμενη από πρόγραμμα

```
program/module interruptCP;
  const
    max = ...;              {μέγιστο μήκος εξόδου}
  type
    text = array [1..max] of char;
  var
    datal, data2 : text;  {απομονωτές δεδομένων}
    outdone: boolean;  {σημαία: τελείωσε η εκτύπωση}
  procedure compute(var comdata: text; var count: integer);
    ...
  procedure start_output(outdata: text; outcount: integer);
    var
      outindex: integer;
    process print;
      begin
        while true do
          begin
            {await_printer_interrupt}
            save_context;
            ioport_dataout := outdata[outindex];
            outindex := outindex + 1;
            outcount := outcount - 1;
            if outcount = 0
              then
                begin
                  disable_interrupts;
                  outdone := true
                end;
            restore_context
          end {while}
      end; {print}
    begin {start_output}
      outindex := 1;
      outdone := false;
      enable_interrupts
    end; {start_output}
  begin {interruptCP}
    disable_interrupts;
    initialize_hardware;
    connect_interrupt_vector(print);
    compute(datal, numl);                  {υπολογισμός1}
    start_output(datal, numl);             {ξεκινά εκτύπωση1, και}
    compute(data2, num2);                  {ταυτόχρονα υπολογισμός2}
    while not outdone do {keeptesting};    {περίμενε μέχρι να τυπωθεί το 1}
    start_output(data2, num2);             {ξεκινά εκτύπωση2, και}
    compute(datal, numl);                  {ταυτόχρονα υπολογισμός3}
    while not outdone do {keeptesting};    {περίμενε μέχρι να τυπωθεί το 2}
    start_output(datal, numl);             {ξεκινά εκτύπωση3, και}
    compute(data2, numl);                  {ταυτόχρονα υπολογισμός4}
    ...
  end {interruptCP}
```

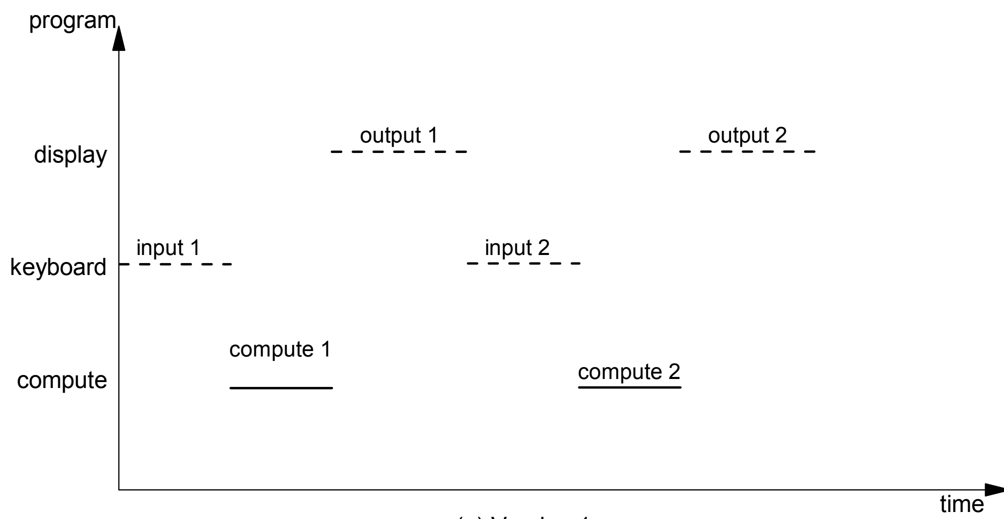
Σχήμα 5.2 Έξοδος ελεγχόμενη από διακοπές

```
program/module concurrent I/O;

    const
        max = 80;               {μέγιστο μήκος εισόδου}
        line_end = return;      {χαρακτήρας τερματισμού γραμμής}
    type
        text = array [1..max] of char;
    var
        indata, outdata, save_data: text;     {data buffers}
        indone, outdone: boolean;             {flags τερματισμού}
        inindex, incount, outindex, outcount, save_count: integer;

procedure process_data;

procedure start_input;

    process keyboard;
        begin
            while true do
                begin
                    {await_kbd_interrupt}
                    save_context;
                    kbd_status := ioport_status;
                    kbd_data := ioport_datain;    {επιβεβαίωση interrupt}
                    if kbd status = OK
                        then                      {λήψη χωρίς σφάλματα}
                            begin
                                indata[inindex] := kbd_data;
                                inindex := inindex + 1;
                                incount:= incount + 1;
                                if kbd data = line_end
                                    then          {η γραμμή εισόδου συμπληρώθηκε}
                                        begin
                                            ioport_command := kbd_interrupts_disable;
                                            indone := true
                                        end {if}
                            end {if}
                        else handle_error(kbd_status);   {σφάλμα στη λήψη}
                    restore_context
                end {while}
        end; {keyboard}

    begin {start_input}
        indone := false;
        inindex := 1;
        incount:= 0;
        ioport_command := kbd_interrupts_enable
    end; {start_input}
```

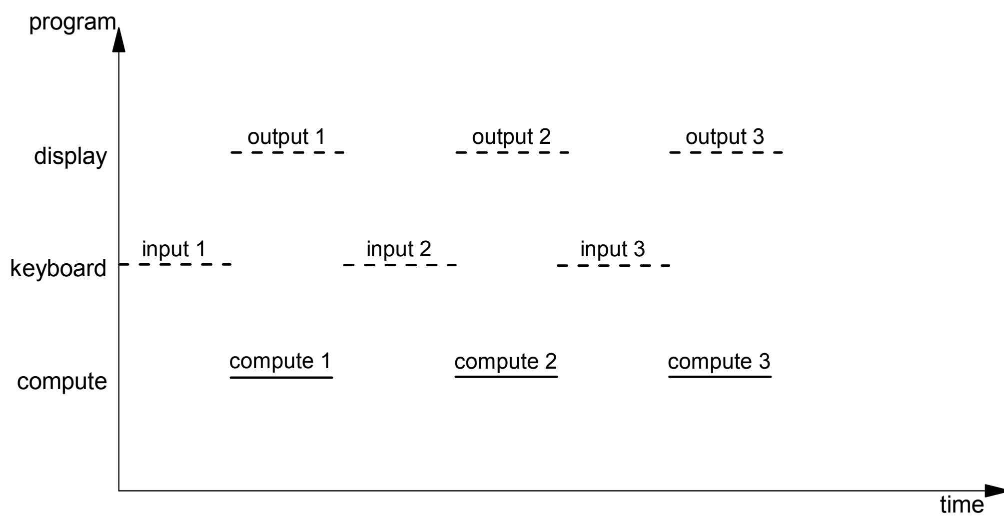
```
procedure start_output;

    process display;
        begin
            while true do
                begin
                    {await_dsp_interrupt}
                    save_context;
                    ioport_dataout := outdata[outindex];
                    outindex:= outindex + 1;
                    outcount := outcount - 1;
                    if outcount = 0
                        then            {η γραμμή τυπώθηκε}
                            begin
                                ioport_command := dsp _interrupts_disable;
                                outdone := true
                            end; {if}
                    restore context
                end {while}
        end; {display}

    begin {start_output}
        outdone := false;
        outindex := 1;
        outcount := save_count;
        ioport_command := dsp _interrupts_enable
    end; {start_output}

{MAIN - version 1}
begin {concurrent I/O}
    ioport_command := kbd_and_dsp_interrupts_disable;
    initialize_hardware;
    connect_interrupt_vectors(kbd, dsp);
    while true do                                   {για πάντα}
        begin
            start_input;                            {ενεργοποίηση εισόδου(x)}
            while not indone do {keeptesting};      {αναμονή τερματισμού}
            copy(indata, outdata);
            save_count := incount;
            process_data;                           {επεξεργασία εισόδου(x)}
            start_output;                           {προβολή εισόδου(x)}
            while not outdone do {keeptesting}
        end {while}
end {concurrent I/O, version 1}
```

```
{MAIN - version 2}
begin {concurrent I/O}
    ioport_command := kbd_and_dsp_interrupts_disable;
    initialize_hardware;
    connect_interrupt_vectors(kbd, dsp);
    while true do
        begin
            start_input;                          {ενεργοποίηση εισόδου(x)}
            while not indone do {keeptesting};    {αναμονή τερματισμού}
            copy(indata, outdata);
            save_count := in count;
            start_output;                         {προβολή εισόδου(x), και}
            process_data;                         {ταυτόχρονη επεξεργασία της}
            while not outdone do {keeptesting}
        end {while}
end {concurrent I/O, version 2}

{MAIN - version 3}
begin (concurrent I/O}
    ioport_command := kbd_and_dsp _interrupts_disable;
    initialize_hardware;
    connect_interrupt_vectors(kbd, dsp);
    outdone := true;
    start_input;                                  {ενεργοποίηση εισόδου(1)}
    while true do
        begin
            while not indone do {keeptesting};    {αναμονή εισόδου(x)}
            copy(indata, save_data);              {αποθήκευση εισόδου(x)}
            save_count := incount;
            start_input;                          {ενεργοποίηση εισόδου(x+1)}
            while not outdone do {keeptesting};   {αναμονή εξόδου(x-1)}
            copy(save_data, outdata);             {είσοδος(x) στην έξοδο}
            start_output;                         {προβολή εισόδου(x), και}
            process_data                          {ταυτόχρονη επεξεργασία της}
        end {while}
end {concurrent I/O, version 3}
```

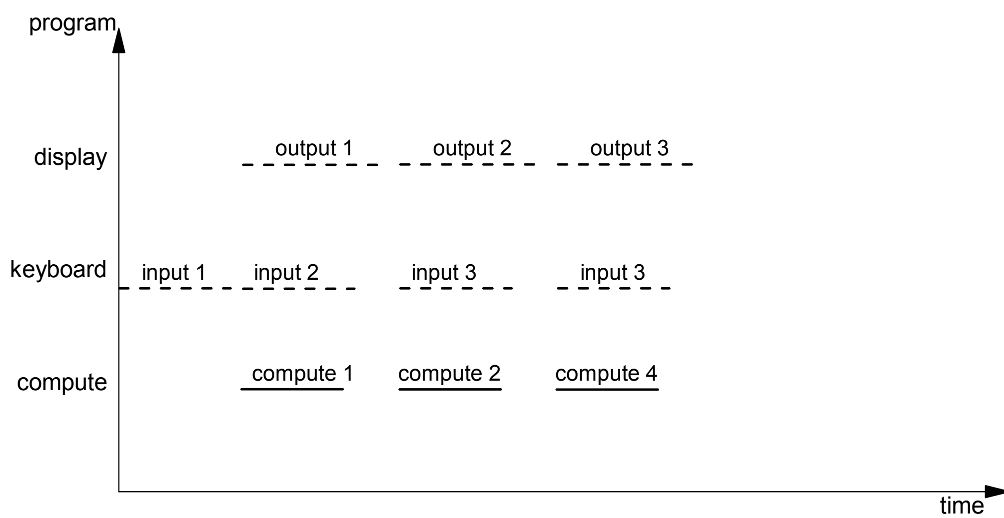
Σχήμα 5.3 Προγραμματισμός I/O με διακοπές.




(a) Version 1




(b) Version 2




(c) Version 3

Σχήμα 5.4 Χρονοδιάγραμμα προγραμμάτων ελέγχου I/O με διακοπές.