ΠΕΡΙΕΧΟΜΕΝΑ

Πρόλογος **11**

Ευχαριστίες **15**

1 Τι είναι ταυτόχρονος προγραμματισμός **17**

1.1 Από το σειριακό στον ταυτόχρονο προγραμματισμό 17
1.2 Ταυτόχρονος προγραμματισμός 20
1.3 Ορθότητα των ταυτόχρονων προγραμμάτων 22
1.4 Πλεκτή ροή 24
1.5 Η προέλευση των λειτουργικών συστημάτων 25
1.6 Λειτουργικά συστήματα και ταυτόχρονος προγραμματισμός 27
1.7 Γενική επισκόπηση του βιβλίου 29
1.8 Συμβολισμός των προγραμμάτων 31
1.9 Ασκήσεις 34

2 Η αφαιρετική θεώρηση του ταυτόχρονου προγραμματισμού **37**

2.1 Εισαγωγή 37
2.2 Αμοιβαίος αποκλεισμός 38
2.3 Ορθότητα 39
2.4 Χρονισμός 42
2.5 Υλοποιηση πρωτογενών εντολών 44
2.6 Ταυτόχρονος προγραμματισμός στην Pascal-s 46
2.7 Περίληψη 47
2.8 Ασκήσεις 48

3 Το πρόβλημα του αμοιβαίου αποκλεισμού **49**

3.1 Εισαγωγή 49
3.2 Πρώτη προσπάθεια 50
3.3 Δεύτερη προσπάθεια 52
3.4 Τρίτη προσπάθεια 54
3.5 Τέταρτη προσπάθεια 56

3.6 Ο αλγόριθμος του Dekker 58
3.7 Απόδειξη του αλγορίθμου του Dekker 61
3.8 Συμπέρασμα 64
3.9 Ασκήσεις 64

4 Σηματοφόροι 71

4.1 Εισαγωγή 71
4.2 Αμοιβαιός αποκλεισμός 72
4.3 Το πρόβλημα του παραγωγού και του καταναλωτή 77
4.4 Και πάλι για το πρόβλημα του παραγωγού και του καταναλωτή 80
4.5 Ο κοιμώμενος κουρέας 84
4.6 Η περιοχή προσωρινής αποθήκευσης περιορισμένου μεγέθους 88
4.7 Ασκήσεις 91

5 Μόνιτορ 97

5.1 Εισαγωγή 97
5.2 Ορισμός των μόνιτορ 99
5.3 Προσομοίωση του σηματοφόρου 103
5.4 Το πρόβλημα των αναγνωστών και των εγγραφέων 105
5.5 Απόδειξη των ιδιοτήτων των μόνιτορ 109
5.6 Η προσομοίωση των μόνιτορ με σηματοφόρους 111
5.7 Μη περιορισμένα σήματα 114
5.8 Ασκήσεις 116

6 Τα ραντεβού της Ada 119

6.1 Εισαγωγή 119
6.2 Η εντολή Accept 120
6.3 Η εντολή Select 126
6.4 Απόδειξη των ιδιοτήτων των ραντεβού 132
6.6 Ασκήσεις 133

7 Το δείπνο των φιλοσόφων 137

7.1 Εισαγωγή 137
7.2 Πρώτη προσπάθεια 138
7.3 Δεύτερη προσπάθεια 139
7.4 Μία σωστή λύση 141
7.5 Κρίσιμες περιοχές υπό συνθήκη 143
7.6 Ασκήσεις 146

Παράρτημα: Κιτ υλοποίησης **149**

Α.1 Εισαγωγή 149
Α.2 Ο μεταγλωττιστής 150
Α.3 Ο κώδικας p 151
Α.4 Κλήσεις διαδικασιών 155
Α.5 Ταυτοχρονισμός 160
Α.6 Σηματοφόροι 164
Α.7 Τυχαιοποίηση 164
Α.8 Λίστα προγράμματος 164

Βιβλιογραφία **195**

Εγχειρίδια 195
Πηγές 195

Ευρετήριο **199**

2 Η ΑΦΑΙΡΕΤΙΚΗ ΘΕΩΡΗΣΗ ΤΟΥ ΤΑΥΤΟΧΡΟΝΟΥ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

2.1 ΕΙΣΑΓΩΓΗ

Ο ταυτόχρονος προγραμματισμός δεν είναι η μελέτη των λειτουργικών συστημάτων ή των συστημάτων πραγματικού χρόνου, αλλά των αφαιρετικών προβλημάτων προγραμματισμού που τίθενται κάτω από ορισμένες προϋποθέσεις. Ο ταυτόχρονος προγραμματισμός αναπτύχθηκε εξαιτίας των προβλημάτων της κατασκευής λειτουργικών συστημάτων, και τα παραδείγματά του είναι αφαιρετικές εκδοχές τέτοιων προβλημάτων. Το πιο σημαντικό είναι ότι οι προϋποθέσεις του ταυτόχρονου προγραμματισμού ικανοποιούνται από πολλά συστήματα, και έτσι οι μέθοδοί του μπορούν να εφαρμόζονται σε πραγματικά συστήματα. Τα συστατικά μέρη ενός συστήματος που δεν επιδέχονται τεχνικές του ταυτόχρονου προγραμματισμού θα πρέπει να εντοπίζονται, και να δίνεται ιδιαίτερη προσοχή στη σχεδίαση και την υλοποίησή τους.

Στο Κεφάλαιο 1, δώσαμε τον ορισμό του ταυτόχρονου προγράμματος. Αποτελείται από πολλές σειριακές διεργασίες, των οποίων οι ακολουθίες εκτέλεσης διαπλέκονται. Τα σειριακά προγράμματα δεν είναι εντελώς ανεξάρτητα μεταξύ τους – αν ήταν, δε θα υπήρχε τίποτα να μελετήσουμε. Πρέπει να επικοινωνούν μεταξύ τους, για να συγχρονίζονται ή να ανταλλάσσουν δεδομένα.

Ο πρώτος τρόπος επικοινωνίας που θα εξετάσουμε είναι η *κοινή μνήμη* (common memory). Αυτός ο τρόπος είναι κατάλληλος για τους ψευδοπαράλληλους υπολογιστές εναλλαγής (switched computers), όπου όλες οι διεργασίες εκτελούνται στον ίδιο επεξεργαστή και χρησιμοποιούν την ίδια φυσική μνήμη. Επίσης χρησιμοποιείται σε μερικά γνήσια παράλληλα συστήματα, όπως είναι οι υπολογιστές CDC Cyber όπου, παρότι μία κεντρική μονάδα επεξεργασίας και δέκα περιφερειακοί επεξεργαστές εκτελούν ξεχωριστά προγράμματα συγχρόνως, ο συγχρονισμός επιτυγχάνεται αν βάλουμε τους περιφερειακούς επεξεργαστές να διαβάζουν και να γράφουν στη μνήμη της κεντρικής μονάδας επεξεργασίας. Στη δική μας αφαιρετική προσέγγιση, η κοινή μνήμη θα αναπαρίσταται απλώς από καθολικές μεταβλητές, προσπελάσιμες από όλες τις διεργασίες.

Η κοινή μνήμη μπορεί επίσης να χρησιμοποιηθεί για την αποθήκευση διαδικασιών περιορισμένης πρόσβασης. Η προσπέλαση αυτών των διαδικασιών ανατίθεται ουσιαστικά σε μία διεργασία. Αυτός είναι ο τρόπος με τον οποίο υλοποιήθηκαν τα περισσότερα λειτουργικά συστήματα τρίτης γενιάς. Τα προγράμματα "συστήμα-

τος" μπορούν να καλούνται μόνο από ειδικές εντολές που εξασφαλίζουν ότι μόνο μία διεργασία κάθε φορά εκτελεί ένα πρόγραμμα συστήματος.

Με την εμφάνιση των κατανεμημένων υπολογιστών (distributed computing), δεν είναι πλέον σωστό να θεωρούμε ότι υπάρχει μία κοινή κεντρική μνήμη. Το Κεφάλαιο 5 εξετάζει τον ταυτόχρονο προγραμματισμό με την αποστολή και λήψη σημάτων, αντί για την ανάγνωση και εγγραφή σε μία κοινή μεταβλητή ή την εκτέλεση μίας κοινής διαδικασίας. Ο συγχρονισμός με μεταβίβαση μηνυμάτων έχει εφαρμοστεί σε πολλά πειραματικά συστήματα για υπολογιστές ενός επεξεργαστή, αλλά αυτή η προσέγγιση δεν είχε ευρεία αποδοχή, εξαιτίας ίσως της χαμηλής απόδοσης που έχει η μεταβίβαση μηνυμάτων σε σύγκριση με τα απλούστερα συστήματα. Φυσικά, τα κατανεμημένα συστήματα δεν έχουν άλλη εκλογή.

2.2 ΑΜΟΙΒΑΙΟΣ ΑΠΟΚΛΕΙΣΜΟΣ

Ο αμοιβαίος αποκλεισμός (mutual exclusion) είναι ένα από τα δύο πιο σημαντικά προβλήματα του ταυτόχρονου προγραμματισμού, καθώς είναι η αφαιρετική έκφραση πολλών προβλημάτων συγχρονισμού. Λέμε ότι η δραστηριότητα A_1 της διεργασίας P_1 και η δραστηριότητα A_2 της διεργασίας P_2 πρέπει να αποκλείουν η μία την άλλη αν η εκτέλεση της A_1 δεν επιτρέπεται να επικαλύπτει την εκτέλεση της A_2. Αν οι P_1 και P_2 επιχειρήσουν να εκτελέσουν τις αντίστοιχες δραστηριότητές τους A_i συγχρόνως, τότε θα πρέπει να εξασφαλίσουμε ότι μόνο η μία από τις δύο θα το καταφέρει. Η "ηττημένη" διεργασία πρέπει να μπλοκαριστεί, δηλαδή δεν πρέπει να συνεχίσει μέχρις ότου η "νικήτρια" διεργασία να ολοκληρώσει την εκτέλεση της δραστηριότητάς της A_i.

Το πιο κοινό παράδειγμα για την ανάγκη αμοιβαίου αποκλεισμού στα πραγματικά συστήματα είναι η κατανομή πόρων. Προφανώς, στην ίδια συσκευή ταινίας δεν είναι δυνατόν να τοποθετηθούν ταυτόχρονα δύο ταινίες. Πρέπει να γίνει κάποια πρόβλεψη για να αποφασίζεται σε ποια διεργασία θα διατίθεται μία ελεύθερη συσκευή, και κάποια άλλη πρόβλεψη ώστε να μπλοκάρονται οι διεργασίες που ζητούν μία συσκευή, όταν δεν είναι διαθέσιμη. Υπάρχει μία προφανής λύση: να εκτελείται μόνο μία εργασία κάθε φορά. Όμως, αυτό έρχεται σε αντίθεση με έναν από τους κύριους στόχους του ταυτόχρονου προγραμματισμού – την παράλληλη εκτέλεση πολλών διεργασιών.

Ουσιαστικός ταυτοχρονισμός είναι δυνατός μόνο αν οι διεργασίες είναι χαλαρά συνδεδεμένες μεταξύ τους. Η χαλαρή σύνδεση χαρακτηρίζεται από την ανάγκη για σύντομες και περιστασιακές επικοινωνίες. Η αφαιρετική διατύπωση του προβλήματος του αμοιβαίου αποκλεισμού μπορεί να εκφραστεί με τον εξής τρόπο:

υπόλοιπο
πρωτόκολλο πριν
κρίσιμος τομέας
πρωτόκολλο μετά

Το "*υπόλοιπο*" θεωρούμε ότι αντιπροσωπεύει κάποιο σημαντικό μέρος της επεξεργασίας. Περιστασιακά, για παράδειγμα μετά από την ολοκλήρωση του *υπολοίπου*, η διεργασία χρειάζεται να εισέλθει σε ένα σύντομο *κρίσιμο τομέα* (critical section). Πριν, και ίσως και μετά από τον κρίσιμο τομέα, εκτελούνται κάποιες ακολουθίες εντολών, που ονομάζονται *πρωτόκολλα*. Τα πρωτόκολλα αυτά εξασφαλίζουν ότι ο κρίσιμος τομέας εκτελείται πραγματικά με τρόπο που αποκλείει όλους τους άλλους κρίσιμους τομείς. Φυσικά, όπως ακριβώς ο κρίσιμος τομέας πρέπει να είναι σύντομος σε σχέση με το κύριο πρόγραμμα, για να έχουμε όφελος από τον ταυτοχρονισμό, πρέπει και τα πρωτόκολλα να είναι σχετικά σύντομα. Τα πρωτόκολλα αντιπροσωπεύουν την επιβάρυνση που πληρώνουμε για τον ταυτοχρονισμό. Στην ευνοϊκή περίπτωση που οι κρίσιμοι τομείς και τα πρωτόκολλα είναι αρκετά μικρά, το σημαντικό μέρος της επεξεργασίας που αφαιρετικά ονομάσαμε *υπόλοιπο* μπορεί να εκτελείται με χρονική επικάλυψη, δικαιολογώντας έτσι τη σχεδίαση του συστήματος πολυπρογραμματισμού.

Υπάρχει ένας άλλος πιο σημαντικός λόγος για τον οποίο απαιτείται χαλαρή σύνδεση ανάμεσα στις ταυτόχρονες διεργασίες, και αυτός είναι η εξασφάλιση της αξιοπιστίας. Θέλουμε να είμαστε εξασφαλισμένοι ότι, αν υπάρξει ένα σφάλμα σε μία από τις διεργασίες, δε θα εξελιχθεί σε κατάρρευση του συστήματος. Θα πρέπει επίσης, αν μία μεμονωμένη συσκευή παρουσιάσει αστοχία, να είναι δυνατόν να υποβιβαστεί ομαλά η απόδοση του συστήματος (fail-soft). Θα ήταν παράλογο να έχουμε κατάρρευση του συστήματος μόνο και μόνο επειδή παρουσίασε πρόβλημα μία συσκευή ταινίας.

Θα θέσουμε την αφαιρετική απαίτηση ότι, αν μία διεργασία τερματίσει αντικανονικά έξω από τον κρίσιμο τομέα, δε θα πρέπει να επηρεάζεται καμία άλλη διεργασία. (Από αυτή την άποψη, τα πρωτόκολλα θεωρούνται μέρος του κρίσιμου τομέα.) Καθώς ο κρίσιμος τομέας είναι το σημείο όπου πραγματοποιείται η επικοινωνία, δεν είναι λογικό να απαιτούμε το ίδιο και για τους κρίσιμους τομείς. Θα μπορούσαμε να χρησιμοποιήσουμε την εξής μεταφορά. Αν ένας δρομέας σε έναν αγώνα σκυταλοδρομίας πέσει αφού έχει δώσει τη σκυτάλη, τότε ο αγώνας δε θα πρέπει να επηρεάζεται. Δεν είναι όμως λογικό να θεωρήσουμε ότι ο αγώνας δεν επηρεάστηκε αν η πτώση συνέβη κατά τις κρίσιμες στιγμές της αλλαγής της σκυτάλης.

Ο περιορισμός αυτός δεν είναι ούτε στην πράξη αστήρικτος. Κρίσιμοι τομείς, όπως οι εργασίες εισόδου-εξόδου στο δίσκο, εκτελούνται συνήθως από κοινές ρουτίνες του συστήματος ή από ρουτίνες που προέρχονται από μεταγλωττιστή (compiler), οι οποίες έχουν γραφεί από έναν ικανό προγραμματιστή συστημάτων. Η πιθανότητα σφάλματος λογισμικού σε μία τέτοια ρουτίνα μάλλον θα είναι πολύ μικρότερη από ό,τι σε ένα συνηθισμένο πρόγραμμα.

2.3 ΟΡΘΟΤΗΤΑ

Τι σημαίνει να είναι σωστό ένα ταυτόχρονο πρόγραμμα; Ένα συνηθισμένο πρόγραμμα είναι σωστό αν σταματά και τυπώνει τη "σωστή" απάντηση. Γενικά, θα ανα-

γνωρίσετε μία "σωστή" απάντηση όταν τη δείτε. Αυτό ισχύει και για ορισμένα ταυτόχρονα προγράμματα, όπως το *sortprogram*.

Από την άλλη μεριά, το πιο χαρακτηριστικό γνώρισμα ενός λειτουργικού συστήματος ή συστήματος πραγματικού χρόνου είναι ότι δεν πρέπει ποτέ να σταματά. Ο μόνος τρόπος για να σταματήσετε ένα τυπικό λειτουργικό σύστημα είναι να πατήσετε το κουμπί εκκίνησης στην πρόσοψη του υπολογιστή. Ένα λειτουργικό σύστημα δεν τυπώνει τίποτα δικό του (εκτός από κάποια διαχειριστικά δεδομένα για τα πεπραγμένα και τους λογαριασμούς των χρηστών, που δεν έχουν σχέση με την ουσία της δουλειάς). Έτσι, όταν έχουμε να κάνουμε με λειτουργικά συστήματα, θα πρέπει να αλλάξουμε τις αντιλήψεις μας για το τι σημαίνει η ορθότητα ενός προγράμματος. Καθώς ο ταυτόχρονος προγραμματισμός χρησιμοποιείται κατά κύριο λόγο στα λειτουργικά συστήματα και τα συστήματα πραγματικού χρόνου, αυτό είναι το πεδίο που πρέπει να μελετηθεί στο βιβλίο.

Μία αυτονόητη αλήθεια είναι ότι ένα πρόγραμμα πρέπει να κάνει ό,τι ορίζουν οι προδιαγραφές του. Αυτό δεν παύει να ισχύει στην περίπτωση των ταυτόχρονων προγραμμάτων, αν και οι προδιαγραφές τους είναι ριζικά διαφορετικές από αυτές των σειριακών. Στην αφαιρετική θεώρησή μας, θα ξεχωρίσουμε δύο τύπους ιδιοτήτων ορθότητας: τις ιδιότητες *ασφάλειας* (safety) και τις ιδιότητες *βιωσιμότητας* (liveness).

Οι ιδιότητες ασφάλειας είναι εκείνες που απαιτούνται από το στατικό μέρος των προδιαγραφών του προγράμματος. Συχνά, αυτές είναι οι μόνες απαιτήσεις που αναφέρονται ρητά. Ο αμοιβαίος αποκλεισμός είναι μία ιδιότητα ασφάλειας: η απαίτηση ότι οι κρίσιμοι τομείς αποκλείουν ο ένας τον άλλο είναι απόλυτη, και δεν αλλάζει κατά τη διάρκεια της εκτέλεσης του προγράμματος. Η ιδιότητα ασφάλειας στο πρόβλημα του παραγωγού και του καταναλωτή είναι ότι ο καταναλωτής πρέπει να καταναλώνει κάθε στοιχείο δεδομένων που παράγεται από τον παραγωγό, και ότι αυτό πρέπει να το κάνει με τη σειρά που παράγονται τα δεδομένα.

Οι ιδιότητες ασφάλειας αντιστοιχούν σε αυτό που στη θεωρία των σειριακών προγραμμάτων είναι γνωστό ως *μερική ορθότητα* (partial correctness): αν το πρόγραμμα τερματιστεί, οι απαντήσεις που θα δώσει πρέπει να είναι "σωστές". Οι ιδιότητες ασφάλειας είναι συνήθως σχετικά εύκολο να αποδειχθούν. Απαιτούνται ρητά από τις προδιαγραφές, και τα προγράμματα σχεδιάζονται έτσι ώστε να τις ικανοποιούν. Μπορείτε πάντα να πετύχετε μεγαλύτερη ασφάλεια αφαιρώντας κάποιο μέρος του ταυτοχρονισμού και αφήνοντας περισσότερα τμήματα της διεργασίας να εκτελούνται σειριακά.

Η παραβίαση του αμοιβαίου αποκλεισμού είναι η αιτία των περισσότερων καταρρεύσεων των λειτουργικών συστημάτων. Συχνά εμπλέκεται και η δυναμική κατανομή μνήμης: μπορεί μία διεργασία να πιστεύει ότι γνωρίζει τη θέση κάποιου πίνακα στη μνήμη, ενώ στην πραγματικότητα ο πίνακας έχει μετακινηθεί και η θέση μνήμης του έχει παραχωρηθεί σε άλλη διεργασία.

Σε ένα υπολογιστικό κέντρο, γνωστό στο συγγραφέα, ένας ασυνείδητος προγραμματιστής κατάφερε να εισαγάγει στο σύστημα ένα πρόγραμμα που με μία δια-

ταγή θα έδινε στα προγράμματά του την υψηλότερη προτεραιότητα. Δε γνώριζε όμως ότι απαιτούνταν πρωτόκολλα αμοιβαίου αποκλεισμού, επειδή το μητρώο χρονοπρογραμματισμού δεν ήταν σε σταθερή θέση στη μνήμη. Έτσι, η εκτέλεση του προγράμματός του συχνά έγραφε πάνω σε κάποιο άλλο μητρώο του συστήματος, προκαλώντας κατάρρευση. Καθώς η κατάρρευση συνέβαινε πολλά λεπτά αργότερα, οι αποτυπώσεις των περιεχομένων της μνήμης δεν έδιναν καμία ένδειξη για το τι είχε συμβεί. Μετά από πολλές εβδομάδες, το πρόβλημα λύθηκε, όταν παρατηρήθηκε η παράξενη διαταγή στο αρχείο καταγραφής πεπραγμένων του συστήματος. Αν ένα τέτοιο σφάλμα είχε συμβεί σε ένα κανονικό πρόγραμμα του συστήματος, θα ήταν πολύ πιο δύσκολο να εντοπιστεί.

Η βιωσιμότητα (liveliness), από την άλλη μεριά, ασχολείται με τις δυναμικές ιδιότητες. Αντιστοιχεί στην *ολική ορθότητα* (total correctness) του σειριακού προγραμματισμού: το πρόγραμμα τερματίζεται και η απάντηση είναι "σωστή". Στα ταυτόχρονα συστήματα, βιωσιμότητα σημαίνει ότι, αν κάτι είναι αναμενόμενο να συμβεί, τελικά θα συμβεί. Αν μία διεργασία θέλει να εισέλθει στον κρίσιμο τομέα της, τελικά θα το κάνει. Αν ένας παραγωγός παράγει δεδομένα, τελικά ο καταναλωτής θα τα καταναλώσει.

Η πιο σοβαρή παραβίαση της βιωσιμότητας είναι η καθολική μορφή που είναι γνωστή ως *αδιέξοδο* (deadlock).[†] Αδιέξοδο σημαίνει ότι ο υπολογιστής δεν κάνει πλέον καμία (χρήσιμη) εργασία. Ένας βρόχος που ψάχνει για μία ελεύθερη συσκευή ταινίας όταν καμία συσκευή ταινίας δεν πρόκειται ποτέ να είναι διαθέσιμη δεν είναι χρήσιμη εργασία. Αν όλες οι διεργασίες είναι σε αναστολή ή βρίσκονται σε τέτοιους βρόχους, τότε λέμε ότι ο υπολογιστής βρίσκεται σε αδιέξοδο. Αυτό το "κρέμασμα" του συστήματος είναι καταστροφικό, καθώς σε ένα σύστημα πολυπρογραμματισμού επηρεάζονται όλοι οι χρήστες.

Το παρακάτω σενάριο αδιεξόδου συνέβη πραγματικά σε ένα λειτουργικό σύστημα. Το σύστημα είχε πολλούς επεξεργαστές. Κάθε εργασία έγραφε τα διαχειριστικά της δεδομένα σε μία κοινή περιοχή μνήμης, η οποία γραφόταν σε ένα αρχείο δίσκου όταν γέμιζε. Τη διαδικασία εγγραφής την εκτελούσε όποιος επεξεργαστής ήταν ελεύθερος. Αν όμως όλοι οι επεξεργαστές επιχειρούσαν να γράψουν τα διαχειριστικά τους δεδομένα την ίδια στιγμή, δεν υπήρχε ελεύθερος επεξεργαστής για να γράψει στο δίσκο. Καθώς κανένα πρόγραμμα δεν παραχωρούσε εθελοντικά τον επεξεργαστή του, το σύστημα βρισκόταν σε αδιέξοδο. Μία τυπική λύση στην περίπτωση αυτή είναι να χρησιμοποιήσει κανείς το τερματικό της κονσόλας του υπολογιστή, για να ξεγελάσει μία διεργασία και να την κάνει να πιστέψει ότι η περιοχή μνήμης είναι άδεια. Φυσικά, θα χαθούν κάποια δεδομένα, αλλά αυτό είναι συνήθως προτιμότερο από το να βρεθεί το σύστημα σε αδιέξοδο και να χρειαστεί να ξεκινήσουν από την αρχή όλα τα προγράμματα που εκτελούνται εκείνη τη στιγμή.

[†] Το αδιέξοδο θεωρητικά κατατάσσεται στις ιδιότητες ασφάλειας, επειδή είναι κάτι που δεν πρέπει να συμβαίνει ποτέ. Ωστόσο, η απουσία αδιεξόδου μπορεί να δειχθεί αν αποδείξουμε ότι υπάρχει πάντα τουλάχιστον μία βιώσιμη (live) διεργασία.

Μία τοπική παραβίαση της βιωσιμότητας ονομάζεται *παραγκωνισμός* (lockout) ή ατομική *λιμοκτονία* (starvation). Στην περίπτωση αυτή, υπάρχει πάντα κάποια διεργασία που μπορεί να προχωρήσει, αλλά κάποια συγκεκριμένη διεργασία καθυστερεί επαόριστον. Ο παραγκωνισμός είναι λιγότερο σοβαρός από το αδιέξοδο, καθώς ο υπολογιστής εξακολουθεί να κάνει κάποια (υποθετικά) χρήσιμη εργασία. Από την άλλη μεριά, ο παραγκωνισμός είναι δύσκολο να ανακαλυφθεί και να διορθωθεί, καθώς μπορεί να συμβεί μόνο σε πολύπλοκα σενάρια, όπου κάποιες διεργασίες αθέλητα συνωμοτούν για να αρνηθούν έναν πόρο σε κάποια άλλη άτυχη διεργασία.

Για λόγους που θα αναπτυχθούν στην επόμενη ενότητα, περιορίζουμε τη συζήτησή μας στην ποιοτική βιωσιμότητα, που σημαίνει ότι η λέξη "τελικά" που χρησιμοποιήθηκε στον ορισμό της βιωσιμότητας, έχει ακριβώς αυτή την έννοια: μέσα σε ένα απροσδιόριστο, αλλά πεπερασμένο χρονικό διάστημα.

Ανάμεσα στην πλήρη αδιαφορία για το χρόνο, στην ιδέα της βιωσιμότητας, και το ρητό προσδιορισμό του χρόνου, βρίσκεται η έννοια της *αμεροληψίας* (fairness): μία διεργασία που θέλει να προχωρήσει πρέπει να έχει δίκαιη μεταχείριση σε σχέση με όλες τις άλλες διεργασίες. Η αμεροληψία είναι πιο δύσκολο να οριστεί με ακρίβεια, και θα την αναφέρουμε μόνο περιστασιακά. Ακόμα, πολλά συστήματα είναι συνειδητά μεροληπτικά, με τη σχεδίαση ενός μηχανισμού καθορισμού προτεραιότητας που ευνοεί κάποιες διεργασίες έναντι άλλων.

2.4 ΧΡΟΝΙΣΜΟΣ

Δεν κάνουμε καμία παραδοχή σχετικά με τις απόλυτες ή τις σχετικές ταχύτητες με τις οποίες εκτελούνται οι διεργασίες. Η δήλωση αυτή με την πρώτη ματιά φαίνεται μάλλον σκανδαλώδης. Ωστόσο, αν αυτός ο περιορισμός δεν τηρηθεί στη σχεδίαση των συστημάτων, μπορεί να προκληθούν σοβαρά σφάλματα. Η δήλωση ξαφνιάζει εξαιτίας της απλοϊκής εμμονής που έχουν οι περισσότεροι προγραμματιστές στο ζήτημα της απόδοσης. Σε τελευταία ανάλυση, γνωρίζουμε ότι ένας δίσκος είναι πιο αργός από μία κεντρική μονάδα επεξεργασίας, και είναι ελκυστική η ιδέα να εκμεταλλευτούμε αυτό το γεγονός στη σχεδίαση όχι μόνο της δομής του συστήματος αλλά και των λεπτομερειών του συγχρονισμού.

Ένας λόγος για την παράβλεψη του χρονισμού είναι ότι η διαίσθησή μας δεν είναι σε θέση να αντιμετωπίσει την κλίμακα των μεγεθών που εμπλέκονται. Ένα έτος έχει μόνο γύρω στο μισό εκατομμύριο λεπτά της ώρας, αλλά ένα δευτερόλεπτο έχει ένα εκατομμύριο μικροδευτερόλεπτα (μs). Είναι ανόητο να λέμε ότι η διεργασία P_1 πρέπει να είναι σε θέση να τελειώσει τον κρίσιμο τομέα της προτού η διεργασία P_2 τελειώσει το δικό της μη κρίσιμο τομέα, εκτός αν μία τέτοια δήλωση στηρίζεται σε λεπτομερείς υπολογισμούς.

Ένας δεύτερος λόγος για την παράβλεψη του χρονισμού είναι ότι τα σφάλματα που εξαρτώνται από το χρόνο είναι εξαιρετικά δύσκολο να εντοπιστούν και να διορθωθούν. Οι ποιοτικές αντιμετωπίσεις, όπως αυτές που παρουσιάζονται στο βι-

βλίο, είναι ανεπαρκείς· πρέπει να συμπεριληφθούν υπολογισμοί χρονισμού. Κάθε τέτοιο σφάλμα μπορεί να σημαίνει εβδομάδες δουλειάς για μία ομάδα προγραμματιστών. Δεχόμαστε λοιπόν οποιοδήποτε λογικό κόστος ως προς την απόδοση, προκειμένου να έχουμε ένα αξιόπιστο σύστημα. Για να αξίζει τον κόπο μία κατάρρευση συστήματος της μισής ώρας, θα πρέπει να έχουμε εξοικονομήσει έναν τεράστιο αριθμό μικροδευτερολέπτων.

Τέλος, υπάρχει ένας πολύ σημαντικός πρακτικός λόγος για να σχεδιάσουμε ένα σύστημα λογισμικού που είναι ανεξάρτητο από παραδοχές χρονισμού, και αυτός είναι η δυναμική φύση της σύνθεσης ενός υπολογιστή. Ακόμα και αν κάποιος είναι σε θέση να εκτελέσει τους αναγκαίους υπολογισμούς για τη βεβαίωση της αξιοπιστίας ενός χρονικά εξαρτημένου συστήματος, είναι στο έλεος οποιασδήποτε αλλαγής στη σύνθεση του συστήματος. Θεωρητικά, η πρόσθεση ενός και μόνο τερματικού θα μπορούσε να αναιρέσει την αξιοπιστία του συστήματος. Αυτό δεν είναι καθόλου υπερβολή. Ένας κατασκευαστής υπολογιστών ξεκίνησε κάποτε τη διάθεση ενός εκσυγχρονισμένου εξαρτήματος που λειτουργούσε σε διπλάσια ταχύτητα από το αρχικό εξάρτημα. Οι πρώτοι πελάτες έλαβαν ένα σημείωμα που ουσιαστικά έλεγε: "Σας ευχαριστούμε που αγοράσατε το διπλής ταχύτητας εξάρτημά μας, αλλά παρακαλούμε να το δουλεύετε στην παλιά ταχύτητα για λίγους μήνες, όσο εμείς ελέγχουμε διεξοδικά το λειτουργικό σύστημα για χρονικές εξαρτήσεις."! Ένα σύστημα μπορεί βέβαια να απαιτεί κάποιες χρονικά εξαρτημένες λειτουργίες, αλλά αυτές οι λειτουργίες θα πρέπει να είναι σαφώς προσδιορισμένες και απομονωμένες, ώστε να εξασφαλίζεται η ευελιξία και η αξιοπιστία του συστήματος.

Τα πλεονεκτήματα που συγκεντρώνει ένα χρονικά ανεξάρτητο (ασύγχρονο) σύστημα μπορούν να επιδειχθούν στο επίπεδο του υλικού με την αρχιτεκτονική των υπολογιστών PDP-11 της DEC. Αντί να συγχρονίζεται η προσπέλαση της μνήμης και των συσκευών εισόδου-εξόδου με ένα ρολόι, όλα τα δεδομένα στο βασικό PDP-11 μεταφέρονται ασύγχρονα, σύμφωνα με ένα πρωτόκολλο που δεν είναι στη σύλληψή του πολύ διαφορετικό από αυτά που θα εξετάσουμε εμείς. Η επιβάρυνση που προκαλείται από αυτό το πρωτόκολλο είναι ότι, για οποιαδήποτε δεδομένη ηλεκτρονική τεχνολογία, το PDP-11 θα είναι λίγο πιο αργό από έναν υπολογιστή που έχει κατασκευαστεί με σύγχρονη (χρονικά εξαρτημένη) σχεδίαση. Βέβαια, η αρχιτεκτονική έχει αποδειχθεί εξαιρετικά ευέλικτη: για παράδειγμα, μονάδες μνήμης οποιωνδήποτε ταχυτήτων μπορούν να συνυπάρχουν ελεύθερα. Αν μία νέα τεχνολογία παράγει μνήμη 1,267 φορές πιο γρήγορη από πριν, μία τέτοια μονάδα μνήμης μπορεί να προστεθεί στο υπάρχον σύστημα, και θα ανταποκρίνεται στο πρωτόκολλο 1,267 φορές ταχύτερα. Ένα σύγχρονο σύστημα μπορεί γενικά να δέχεται τη συνύπαρξη μόνο μονάδων μνήμης που οι ταχύτητές τους είναι πολλαπλάσια η μία της άλλης.

Στην αφαιρετική προσέγγιση, θα γίνουν κάποιες παραδοχές ώστε να αποφύγουμε ανούσιες ακραίες περιπτώσεις. Καθώς η επαόριστον καθυστέρηση δε διαφέρει από το σταμάτημα, θα δεχθούμε (καθολικά) ότι, αν υπάρχει τουλάχιστον μία διεργασία έτοιμη να εκτελεστεί, τότε οπωσδήποτε επιτρέπεται σε κάποια (μη καθοριζόμενη) διεργασία να εκτελεστεί σε πεπερασμένο χρόνο. Δεχόμαστε επίσης (τοπι-

κά) ότι, αν επιτραπεί σε μία διεργασία να μπει στον κρίσιμο τομέα της, θα ολοκληρώσει την εκτέλεση του κρίσιμου τομέα σε ένα πεπερασμένο χρονικό διάστημα.

Από την άλλη μεριά, μπορούμε να χρησιμοποιήσουμε την "αντίπαλη" προσέγγιση για τον έλεγχο των πιθανών σφαλμάτων. Ένα ταυτόχρονο πρόγραμμα πάσχει από αδιέξοδο, αν είναι δυνατόν να επινοήσει κανείς ένα σενάριο αδιεξόδου μόνο με βάση τις παραδοχές του πεπερασμένου, που είδαμε στην προηγουμένη παράγραφο. Αν κάποιος σάς προσφέρει ένα ταυτόχρονο πρόγραμμα, μπορείτε να προσαρμόσετε το αντί-σενάριο ειδικά για το συγκεκριμένο πρόγραμμα. Είστε ένας "αντίπαλος", που επιτρέπεται να μηχανορραφεί ενάντια στο πρόγραμμα.

2.5 ΥΛΟΠΟΙΗΣΗ ΠΡΩΤΟΓΕΝΩΝ ΕΝΤΟΛΩΝ

Οι λύσεις μας στο πρόβλημα του αμοιβαίου αποκλεισμού πάντα "κόβουν δρόμο", κάνοντας χρήση των λειτουργιών αμοιβαίου αποκλεισμού που είναι διαθέσιμες σε ένα κατώτερο επίπεδο – στο επίπεδο του υλικού. Όπως ακριβώς ο χρήστης μίας γλώσσας υψηλού επιπέδου δε χρειάζεται να γνωρίζει πώς λειτουργεί ένας μεταγλωττιστής, εφόσον έχει στη διάθεσή του μία ακριβή περιγραφή της σύνταξης και της σημασιολογίας της γλώσσας, έτσι και εμείς δε θα ασχοληθούμε με τον τρόπο υλοποίησης του υλικού, εφόσον έχουμε στη διάθεσή μας μία ακριβή περιγραφή της σύνταξης και της σημασιολογίας της αρχιτεκτονικής. Κατ' επέκταση, το ίδιο γίνεται και σε χαμηλότερα επίπεδα – ο σχεδιαστής της λογικής του υπολογιστή δε χρειάζεται να γνωρίζει ακριβώς τον τρόπο υλοποίησης ενός ολοκληρωμένου κυκλώματος. Ο σχεδιαστής ενός ολοκληρωμένου κυκλώματος χρειάζεται μόνο να ασχολείται με τις ηλεκτρονικές ιδιότητες των ημιαγωγών και δε χρειάζεται να γνωρίζει όλες τις λεπτομέρειες της κβαντικής φυσικής που τις εξηγούν.

Στα συστήματα κοινής μνήμης, υπάρχει ένας *διαιτητής* (arbiter), που φροντίζει για τον αμοιβαίο αποκλεισμό κατά την προσπέλαση μίας μεμονωμένης λέξης της μνήμης. Η λέξη "προσπέλαση" είναι ένας γενικός όρος για την ανάγνωση και την εγγραφή ή, όπως ονομάζονται συνήθως, τη φόρτωση και την αποθήκευση, σε αντιστοιχία με τις εντολές Load και Store του συμβολομεταφραστή (assembler) γι' αυτές τις ενέργειες. Ο διαιτητής εξασφαλίζει ότι, σε περίπτωση χρονικής σύμπτωσης μεταξύ προσπελάσεων, ο αμοιβαίος αποκλεισμός θα επιτευχθεί με τη διαδοχική εκτέλεση των προσπελάσεων. Για τον προγραμματιστή, η σειρά με την οποία θα γίνουν οι προσπελάσεις δεν είναι εγγυημένη. Ωστόσο, είναι εξασφαλισμένη η λογική συνέπεια των προσπελάσεων, όπως περιγράφηκε στο Κεφάλαιο 1.

Σημειώστε ότι η προσπέλαση μίας μεμονωμένης λέξης είναι μία ενέργεια που μπορεί να μην εμφανίζεται σε μία γλώσσα υψηλού επιπέδου. Ας υποθέσουμε ότι το n είναι μία καθολική μεταβλητή που αρχικά έχει την τιμή μηδέν και χρησιμοποιείται ως μετρητής από πολλές διεργασίες οι οποίες εκτελούν την εντολή $n := n + 1$. Ο μεταγλωττιστής μεταγλωττίζει μία τέτοια εντολή σε τρεις εντολές του συμβολομεταφραστή:

Load	n
Add	1
Store	n

Φανταστείτε τώρα το εξής σενάριο: Η τιμή του n είναι 6. Η P_1 εκτελεί τη Load n και μετά η P_2 επίσης εκτελεί τη Load n. Η P_1 αυξάνει την τιμή του n στον εσωτερικό καταχωρητή της σε 7. Παρόμοια και η P_1 αυξάνει την τιμή στο δικό της εσωτερικό καταχωρητή σε 7. Τέλος, οι δύο διεργασίες εκτελούν την εντολή Store διαδοχικά, και η τιμή 7 αποθηκεύεται δύο φορές. Έτσι, η τελική τιμή του n είναι 7. Δηλαδή, αυξήσαμε την τιμή 6 κατά 1 δύο φορές, και πήραμε 7.

Διαιτητές κοινής μνήμης υπάρχουν και στα πολυεπεξεργαστικά συστήματα και στα μονοεπεξεργαστικά συστήματα των οποίων οι συσκευές εισόδου-εξόδου είναι διευθετημένες για *άμεση προσπέλαση μνήμης* (direct memory access, DMA). Κανονικά, μία συσκευή εισόδου-εξόδου θα μετέφερε κάθε λέξη δεδομένων στην κεντρική μονάδα επεξεργασίας, για να την αποθηκεύσει στη μνήμη. Όμως, αυτό επιβαρύνει απαράδεκτα την κεντρική μονάδα επεξεργασίας. Αντί να γίνει αυτό λοιπόν, δίνεται στη συσκευή εισόδου-εξόδου η διεύθυνση ενός μπλοκ μνήμης. Η συσκευή διακόπτει την κεντρική μονάδα επεξεργασίας μόνο όταν έχει ολοκληρωθεί η μεταφορά του μπλοκ. Υπάρχει ένας διαιτητής που εξασφαλίζει ότι κάθε φορά μόνο μία συσκευή (ή η κεντρική μονάδα επεξεργασίας) μπορεί να προσπελάσει τη μνήμη.

Στην περίπτωση αυτή, λέμε ότι η άμεση προσπέλαση μνήμης πραγματοποιείται με *κλοπή κύκλου* (cycle stealing). Θεωρούμε ότι η μνήμη οδηγείται με τη μέγιστη ταχύτητα προσπέλασης – ας πούμε με μία προσπέλαση ανά millisecond. Κάθε τέτοια προσπέλαση ονομάζεται επίσης κύκλος μνήμης. Για την υλοποίηση της άμεσης προσπέλασης μνήμης, επιτρέπεται κανονικά στην κεντρική μονάδα επεξεργασίας να κάνει υπολογισμούς και να προσπελάζει τη μνήμη. Όταν μία λέξη δεδομένων φτάσει από μία συσκευή εισόδου-εξόδου, αφαιρείται από την κεντρική μονάδα επεξεργασίας το δικαίωμα να προσπελάζει τη μνήμη και επιτρέπεται στη συσκευή να "κλέψει" έναν κύκλο μνήμης. Δεν υπάρχει στην πραγματικότητα καμία επιβάρυνση. Ο κύκλος μνήμης χρειάζεται έτσι κι αλλιώς για την αποθήκευση της λέξης, και με την κλοπή κύκλων η κεντρική μονάδα επεξεργασίας δε χρειάζεται να απασχολείται με μεμονωμένες λέξεις.

Εμπιστευόμαστε τη σωστή λειτουργία του υλικού του υπολογιστή. Ασχολούμαστε μόνο με την ορθότητα του λογισμικού του συστήματος. Αυτό φυσικά δεν ισχύει πάντα, και στην πράξη πρέπει κανείς να προσέχει για δυσλειτουργίες του υλικού. Ένα από τα πιο εντυπωσιακά σφάλματα που γνωρίζει ο συγγραφέας προκλήθηκε από μία αστοχία του υλικού που είχε ως αποτέλεσμα την ανάμιξη δύο διευθύνσεων μνήμης αντί για την εναλλασσόμενη χρήση τους. Το τελικό αποτέλεσμα ήταν μία αποθήκευση δεδομένων στη διεύθυνση που προέκυπτε από την ανάμιξη, και οι ειδικοί του λογισμικού δεν κατάφεραν ποτέ να εξηγήσουν την παρουσία ξένων δεδομένων σε αυτές τις διευθύνσεις μνήμης. Ευτυχώς, τέτοια πράγματα συμβαίνουν σπάνια.

Ένας άλλος τρόπος χρήσης ενός συστήματος κοινής μνήμης είναι ο ορισμός μίας πρωτογενούς κλήσης διαδικασίας, που εγγυάται τον αποκλεισμό των άλλων κλήσεων της ίδιας διαδικασίας. Δηλαδή, αν δύο διεργασίες επιχειρήσουν να καλέσουν την ίδια διαδικασία, μόνο η μία θα το καταφέρει και η "ηττημένη" διεργασία θα πρέπει να περιμένει. Όπως συνήθως, δεν προσδιορίζεται με ποια σειρά θα εξυπηρετούνται οι ταυτόχρονες κλήσεις.

Στα συστήματα πολυπρογραμματισμού, χρησιμοποιείται ο μηχανισμός των διακοπών (interrupts). Μία κρίσιμη διαδικασία γράφεται ως ρουτίνα διακοπής, η οποία θα εκτελείται όταν μία διεργασία προκαλεί διακοπή. Μία σημαία υλικού εξασφαλίζει τον αμοιβαίο αποκλεισμό εμποδίζοντας τη διακοπή – θέτει τον υπολογιστή σε μία κατάσταση που δε δέχεται διακοπές. Με την ολοκλήρωση της ρουτίνας της διακοπής, η σημαία επαναφέρεται και μπορεί τώρα μία άλλη διεργασία να προκαλέσει διακοπή.

Μία άλλη μέθοδος υλοποίησης του αμοιβαίου αποκλεισμού είναι η περιόδευση (polling). Κάθε διεργασία ερωτάται με τη σειρά της αν χρειάζεται κάποια υπηρεσία που πρέπει να πραγματοποιηθεί κάτω από συνθήκες αμοιβαίου αποκλεισμού.

Θα επιτρέψουμε στον εαυτό μας την πολυτέλεια να ορίσουμε πρωτογενείς εντολές, και δε θα μας απασχολήσει η υλοποίηση, πέρα από την επιγραμματική παρουσίασή της στην ενότητα αυτή. Δε θα είναι πολύ δύσκολο κάποιος με λίγη πείρα στην αρχιτεκτονική υπολογιστών και τις δομές δεδομένων να υλοποιήσει οποιοδήποτε από αυτά τα πρωτογενή στοιχεία. Ωστόσο, οι λεπτομέρειες διαφέρουν πάρα πολύ από υπολογιστή σε υπολογιστή. Η μελέτη του κιτ υλοποίησης μπορεί να βοηθήσει. Στη βιβλιογραφία, παραπέμπουμε σε αρκετές περιγραφές υλοποιήσεων ταυτόχρονου προγραμματισμού. Ακόμα, ο φοιτητής που ενδιαφέρεται σοβαρά θα πρέπει να μελετήσει την αρχιτεκτονική του υπολογιστή και του λειτουργικού συστήματος που χρησιμοποιεί.

2.6 ΤΑΥΤΟΧΡΟΝΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΣΤΗΝ PASCAL-S

Η σειριακή Pascal (και το υποσύνολό της που χρησιμοποιείται σε αυτό το βιβλίο) πρέπει να ενισχυθεί με έννοιες του ταυτόχρονου προγραμματισμού. Οι ταυτόχρονες διεργασίες είναι γραμμένες ως διαδικασίες Pascal, και το ότι είναι ταυτόχρονες διεργασίες υποδηλώνεται από την τοποθέτησή τους μέσα στην εντολή **cobegin** ... **coend**.

cobegin P_1; P_2; ...; P_n **coend**.

Μία αίτηση για ταυτόχρονη εκτέλεση πολλών διεργασιών μπορεί να εμφανίζεται μόνο στο κύριο πρόγραμμα, και δεν μπορεί να είναι ένθετη. Η σημασία της εντολής **cobegin** ... **coend** ορίζεται ως εξής:

Η εντολή **cobegin** είναι ένα σήμα προς το σύστημα ότι οι περικλειόμενες διαδικασίες δεν πρέπει να εκτελεστούν, αλλά πρέπει να σημειωθούν για ταυτόχρονη εκτέλεση. Όταν φτάσουμε στην εντολή **coend**, διακόπτεται η εκτέλεση του κύριου προγράμματος και εκτελούνται οι ταυτόχρονες διεργασίες. Η πλεκτή εκτέλεση αυτών των διεργασιών δεν είναι προβλέψιμη, και μπορεί να αλλάζει κάθε φορά. Όταν ολοκληρωθούν όλες οι ταυτόχρονες διεργασίες, συνεχίζεται η εκτέλεση του κύριου προγράμματος με την επόμενη από την **coend** εντολή.

Μία πρόσθετη συμβολική επινόηση που χρησιμοποιούμε είναι η εντολή **repeat** ... **forever** που είναι ακριβώς ισοδύναμη, όσον αφορά το σημασιολογικό της περιεχόμενο, με τη **repeat** ... **until** *false*. Όμως, αυτή η τελευταία είναι κάπως στρυφνή, γι' αυτό προτιμάμε τον πιο ξεκάθαρο συμβολισμό.

Η χρήση της **repeat** ... **forever** τονίζει το γεγονός ότι αυτά τα παραδείγματα προορίζονται για πρωτότυπα κυκλικών προγραμμάτων σε λειτουργικά συστήματα και σε συστήματα πραγματικού χρόνου.

Για να εκτελέσετε οποιοδήποτε από τα παραδείγματα του βιβλίου με το κιτ υλοποίησης, σε γενικές γραμμές θα πρέπει να κάνετε τα παρακάτω. (Δείτε στην Εικόνα 1.5, και για ένα μεγαλύτερο παράδειγμα στην Εικόνα 4.18.)

1. Αντικαταστήστε τη **repeat** ... **forever** με ένα βρόχο **for** που θα εκτελεί την κάθε διεργασία ένα σταθερό αριθμό φορών ή καθορίστε κάποιον άλλο τρόπο τερματισμού της.
2. Παρεμβάλετε εντολές *write* στο κύριο πρόγραμμα ή στις ταυτόχρονες διεργασίες, για να παρακολουθείτε την εκτέλεση του προγράμματος.
3. Σε πολλές περιπτώσεις, κάποιες διαδικασίες δεν έχουν οριστεί, ώστε να τονιστεί η γενικότητα των αλγορίθμων. Για παράδειγμα, στο πρόβλημα του παραγωγού και του καταναλωτή καλούμε τις διαδικασίες *produce* και *consume*. Αυτές οι διαδικασίες πρέπει να οριστούν. Μία δυνατότητα είναι η *produce* να αυξάνει έναν ακέραιο και η *consume* να τον καταναλώνει τυπώνοντάς τον.
4. *Προσοχή* Ο ερμηνευτής του κιτ είναι πολύ ανεπαρκής, γι' αυτό μην το παρακάνετε με το μέγεθος των προγραμμάτων που σκοπεύετε να εκτελέσετε.

2.7 ΠΕΡΙΛΗΨΗ

Ο παρακάτω κατάλογος συνοψίζει την αφαιρετική θεώρηση του ταυτόχρονου προγραμματισμού.

1. Ένα ταυτόχρονο πρόγραμμα αποτελείται από δύο ή περισσότερα σειριακά προγράμματα, που οι ακολουθίες εκτέλεσής τους διαπλέκονται.
2. Οι διεργασίες πρέπει να είναι χαλαρά συνδεδεμένες. Ειδικότερα, η αποτυχία οποιασδήποτε διεργασίας, πέρα από τον κρίσιμο τομέα της και τα πρωτόκολλά της, δεν πρέπει να επηρεάζει τις άλλες διεργασίες.

3. Ένα ταυτόχρονο πρόγραμμα είναι σωστό αν δεν πάσχει από παραβιάσεις των ιδιοτήτων ασφάλειας, όπως ο αμοιβαίος αποκλεισμός, και των ιδιοτήτων βιωσιμότητας, όπως το αδιέξοδο και ο παραγκωνισμός.
4. Ένα ταυτόχρονο πρόγραμμα είναι λανθασμένο, αν υπάρχει μία διαπλεκόμενη ακολουθία εκτέλεσης που παραβιάζει μία από τις απαιτήσεις ορθότητας. Επομένως, αρκεί η κατασκευή ενός σεναρίου για να δειχθεί η μη ορθότητα. Για να δειχθεί η ορθότητα απαιτείται μία μαθηματική επιχειρηματολογία που να δείχνει ότι το πρόγραμμα είναι σωστό για όλες τις ακολουθίες εκτέλεσης.
5. Δε γίνεται καμία παραδοχή για το χρονισμό, εκτός από το ότι (α) καμία διεργασία δε διακόπτεται στον κρίσιμο τομέα της και (β), αν υπάρχουν διεργασίες έτοιμες για εκτέλεση, μία από αυτές οπωσδήποτε προωθείται για εκτέλεση. Μπορούμε να θέσουμε και πρόσθετες απαιτήσεις αμεροληψίας.
6. Θα επεκτείνουμε τη βασική μας γλώσσα προγραμματισμού με πρωτογενείς εντολές συγχρονισμού. Εφόσον η σύνταξη και η σημασία αυτών των εντολών είναι σαφώς ορισμένες, δε μας απασχολεί ο τρόπος υλοποίησής τους.

2.8 ΑΣΚΗΣΕΙΣ

2.1 *Διαρκής άσκηση* Γράψτε τυπικές προδιαγραφές για τα προγράμματα του βιβλίου.

Παράδειγμα:

Προδιαγραφές για το *sortprogram*.

Είσοδος: Μία ακολουθία 40 ακεραίων: $a = \{a_1, \ldots, a_{40}\}$.

Έξοδος: Μία ακολουθία 40 ακεραίων: $b = \{b_1, \ldots, b_{40}\}$.

Ιδιότητα ασφάλειας: όταν το πρόγραμμα τερματιστεί, τότε (i) το b είναι μία μετάθεση του a και (ii) το b είναι ταξινομημένο, δηλαδή για $1 <= i < 40$, $b_i <= b_i+1$.

Ιδιότητα βιωσιμότητας: το πρόγραμμα τερματίζεται.

2.2 *Διαρκής άσκηση* Δοκιμάστε τα παραδείγματα προγραμμάτων του βιβλίου.