6. Παράλληλος Προγραμματισμός

Σκοπός της σχεδίασης και κατασκευής παράλληλων υπολογιστών είναι η εκτέλεση προγραμμάτων σε μικρότερο χρόνο από εκείνον που θα χρειαζόταν ένας υπολογιστής με ένα μόνο επεξεργαστή της ίδιας τεχνολογίας, δηλαδή ένας μονοεπεξεργαστής (uniprocessor). Η επίτευξη αυτού του στόχου μας εισάγει στην τεχνολογία του παράλληλου προγραμματισμού στην οποία εμφανίζονται ορισμένες νέες έννοιες και κάποια καινούργια προβλήματα που δεν υπάρχουν στον απλό προγραμματισμό του κλασσικού μονοεπεξεργαστή.

Στο κεφάλαιο αυτό θα μελετήσουμε αν είναι δυνατή η απεριόριστη αύξηση της ταχύτητας εκτέλεσης ενός προγράμματος με την απεριόριστη προσθήκη επεξεργαστών. Κατόπιν θα αναλύσουμε κάποιες βασικές λειτουργίες που πρέπει να προσφέρει το παράλληλο λειτουργικό σύστημα για την επικοινωνία μεταξύ των διεργασιών και πώς αυτές μπορούν να χρησιμοποιηθούν από το λογισμικό υψηλού επιπέδου που γράφει ο χρήστης με στόχο την προστασία κοινών περιοχών μνήμης και το συγχρονισμό των διεργασιών. Επίσης τα αδιέξοδα και η αποφυγή τους είναι ένα καίριο ζήτημα στον παράλληλο προγραμματισμό που φυσικά εκλείπει στον απλό προγραμματισμό του μονοεπεξεργαστή. Τα θέματα των αδιεξόδων θα μελετηθούν σε μια ειδική ενότητα. Τέλος ο προγραμματισμός παράλληλων διεργασιών από το χρήστη απαιτεί ειδικές εντολές ή και δομές δεδομένων που προσφέρονται από τις παράλληλες γλώσσες υψηλού επιπέδου. Τέτοιες εντολές είναι π.χ. οι fork/join, parbegin/parend, ενώ η δομή monitor είναι μια εναλλακτική λύση που θα μελετηθεί σε ειδική ενότητα.

6.1 Επικοινωνία μεταξύ διεργασιών

Η επικοινωνία μεταξύ διαφορετικών διεργασιών που εκτελούνται παράλληλα σε διαφορετικούς επεξεργαστές είναι πρόβλημα κεφαλαιώδους σημασίας αφού η καρδιά του παράλληλου προγραμματισμού είναι η ταυτόχρονη εκτέλεση διαφόρων εργασιών που συνεργάζονται στενά για τη λύση ενός προβλήματος. Η επικοινωνία μεταξύ των διεργασιών αυτών είναι απαραίτητη για τρεις λόγους:

α. για την ορθή χρήση κοινών αγαθών (συνήθως κοινών μεταβλητών και κοινών περιοχών μνήμης). Η χρήση τεχνικών όπως οι σηματοφορείς και οι κλειδαριές είναι μέθοδοι που χρησιμοποιούνται σε μηχανές με κοινή μνήμη (πολυεπεξεργαστές). Σε πολυυπολογιστές όπου δεν υπάρχουν κοινές περιοχές μνήμης οι κοινές μεταβλητές γίνονται προσπελάσιμες με ανταλλαγές μηνυμάτων μεταξύ των επεξεργαστών.

β. για το συγχρονισμό μεταξύ των διεργασιών

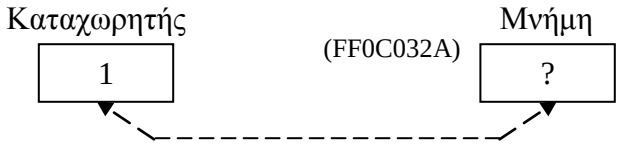
γ. για την ανταλλαγή δεδομένων. Η ανταλλαγή δεδομένων μεταξύ πολυεπεξεργαστών με κοινή μνήμη γίνεται με χρήση κοινών μεταβλητών, ενώ σε πολυυπολογιστές γίνεται με ανταλλαγή μηνυμάτων στο δίκτυο.

Στην ενότητα αυτή θα μελετήσουμε τις τεχνικές που χρησιμοποιούνται για την επικοινωνία μεταξύ διεργασιών που εκτελούνται σε περιβάλλον πολυεπεξεργαστών με κοινή μνήμη. Επειδή οι βασικές λειτουργίες επικοινωνίας που θα δούμε στην ενότητα αυτή είναι πολύ λεπτές και έχουν πολλές ιδιαιτερότητες περιλαμβάνονται στα πλαίσια ενός παράλληλου λειτουργικού συστήματος και σπανίως ο χρήστης έχει τη δυνατότητα να επέμβει σ' αυτές. Οι λειτουργίες ωστόσο που θα περιγραφούν παρακάτω είναι σχεδιασμένες έτσι ώστε να δίνουν στο χρήστη τη δυνατότητα να συνθέσει λογισμικό που να λύνει τα προβλήματα συγχρονισμού και γενικά επικοινωνίας μεταξύ διεργασιών χωρίς να επεμβαίνει στις λεπτομέρειες υλοποίησής τους.

6.1.1 Βασικές λειτουργίες του υλικού

Η επικοινωνία και ο συγχρονισμός των διεργασιών πρέπει να βασίζεται σε κάποιες στοιχειώδεις λειτουργίες που προσφέρει το υλικό (hardware). Αυτές οι λειτουργίες πρέπει να είναι *αδιάσπαστες* ή *ατομικές* (*atomic*) δηλαδή να μην υπάρχει η δυνατότητα να διακοπούν από άλλες εντολές. Για παράδειγμα, μια τέτοια λειτουργία μπορεί να είναι η *ατομική ανάγνωση-και-εγγραφή* στη μνήμη μέσα σε ένα κύκλο μηχανής. Φυσικά μια τέτοια λειτουργία απαιτεί ειδική σχεδίαση αφού συνήθως οι μνήμες απαιτούν ένα κύκλο μόνο για ανάγνωση και άλλον ένα μόνο για εγγραφή.

- **Ατομική ανταλλαγή (atomic exchange).** Η λειτουργία της ατομικής ανάγνωσης-και-εγγραφής επιτρέπει την *ατομική ανταλλαγή* πληροφορίας (*atomic exchange*) μεταξύ ενός καταχωρητή και μιας θέσης μνήμης. Μια τέτοια διαδικασία φαίνεται στο παρακάτω σχήμα:



Το περιεχόμενο του καταχωρητή ανταλλάσσεται με το περιεχόμενο π.χ. της θέσης μνήμης FF0C032A. Η ανταλλαγή εκτελείται μέσα σε ένα μόνο κύκλο έτσι ώστε να μη μπορεί να διακοπεί από κάποια άλλη εντολή.

- **Διάβασε-και-αύξησε (fetch-and-add).** Η λειτουργία αυτή διαβάζει μια μεταβλητή από την μνήμη σε ένα καταχωρητή και αυξάνει την τιμή στη μνήμη κατά 1. Όλα αυτά βέβαια γίνονται αδιάσπαστα σε ένα κύκλο.

- **Έλεγξε-και-Γράψε (Test-and-Set).** Η λειτουργία αυτή διαβάζει μια θέση μνήμης και αν αυτή περιέχει την τιμή 0 τότε γράφει 1, αλλιώς την αφήνει όπως είναι. (Επίσης αδιάσπαστη σε ένα μόνο κύκλο).

Πώς θα μπορούσαν να χρησιμοποιηθούν οι παραπάνω λειτουργίες για να επικοινωνήσουν δύο παράλληλες διεργασίες που εκτελούνται σε ένα πολυεπεξεργαστή; Αυτό γίνεται με χρήση κάποιων απλών τεχνικών του λογισμικού οι οποίες κλειδώνουν κάποιο τμήμα της μνήμης μη επιτρέποντας την πρόσβασή του από άλλες διεργασίες. Το κλείδωμα αυτό γίνεται με διάφορους τρόπους όπως θα δούμε παρακάτω.

6.1.2 Χρήση κοινών αγαθών

Κοινά αγαθά τις περισσότερες φορές είναι οι κοινές μεταβλητές που χρησιμοποιούνται από πολλές διαφορετικές διεργασίες. Για παράδειγμα ένας μετρητής που αυξάνεται κατά ένα από κάθε διεργασία είναι μια κοινή μεταβλητή. Έστω για παράδειγμα ότι έχουμε τις δύο παρακάτω διεργασίες

Διεργασία Α	Διεργασία Β
. . .	. . .
load R1,X	load R1,X
R1 = R1+1	R1 = R1+1
store X,R1	store X,R1
. . .	. . .

Έστω ότι αρχικά η διεύθυνση `X` περιέχει την τιμή 0. Αν και οι δύο διεργασίες διαβάσουν την τιμή του `X` από τη μνήμη πριν η άλλη διεργασία προλάβει να την αλλάξει τότε και οι δύο θα σώσουν στη μνήμη την τιμή `1`. Αν όμως η διεργασία Α, για παράδειγμα, διαβάσει πρώτη την `X` και προλάβει να εκτελέσει τις εντολές `R1 = R1+1` και `store X,R1` πριν η διεργασία Β εκτελέσει `load R1,X` τότε η Β θα διαβάσει την τιμή 1 και θα γράψει την τιμή 2. Κάτι αντίστοιχο θα συμβεί και αν η διεργασία Β γράψει στη μνήμη πριν η διεργασία Α διαβάσει. Το πρόβλημα με τις παραπάνω διεργασίες είναι προφανές. Δεν είναι τόσο ότι μπορεί να πάρουμε λάθος αποτέλεσμα αλλά το ότι δεν ξέρουμε τι αποτέλεσμα θα πάρουμε. Αυτό εξαρτάται από την ταχύτητα ή την προτεραιότητα που έχει η κάθε διεργασία και δεν εξαρτάται από τον προγραμματιστή.

Το πρόβλημα θεραπεύεται με τη χρήση κάποιων μεταβλητών που καλούνται *σηματοφορείς* (*semaphores*) και χρησιμεύουν στο κλείδωμα και το ξεκλείδωμα των κοινών μεταβλητών (shared variables) έτσι ώστε μια μόνο διεργασία να έχει αποκλειστική πρόσβαση σ' αυτές. Το τμήμα του κώδικα μιας διεργασίας στο οποίο γίνεται χρήση μιας κοινής μεταβλητής λέγεται *κρίσιμο τμήμα* (*critical segment*) και προστατεύεται από ένα σηματοφορέα. Οι λειτουργίες του κλειδώματος και ξεκλειδώματος ενός σηματοφορέα ονομάστηκαν `P()` και `V()` από τον Dijkstra ο οποίος έκανε και μια θεωρητική ανάλυση του προβλήματος.

Ένας δυαδικός σηματοφορέας S παίρνει τις τιμές 1 και 0 όπου συνήθως 0 δηλώνει κλείδωμα και 1 δηλώνει ελεύθερη πρόσβαση. Οι λειτουργίες P και V ορίζονται στην περίπτωση αυτή ως εξής:

P(S) /*κλείδωμα*/:

```
while(S==0)
   /*wait and do nothing*/;
S=0;          /*αν S=1 μπες μέσα στο ΚΤ και κλείδωσε*/
```

V(S) /*ξεκλείδωμα*/:

```
S=1;
```

Ο δυαδικός σηματοφορέας χρησιμοποιείται στην περίπτωση που μόνο μια διεργασία επιτρέπεται να βρίσκεται μέσα στο κρίσιμο τμήμα κάθε φορά. Υποτίθεται ότι οι διεργασίες `P` και `V` εκτελούνται αδιάσπαστα. Θα δούμε παρακάτω ότι το `P()` είναι ουσιαστικά μια κλειδαριά με στριφογύρισμα γύρω από τη μεταβλητή `S` (spin-lock).

Οι διεργασίες Α και Β του παραπάνω παραδείγματος τώρα μπορούν να γραφτούν ως εξής

Διεργασία Α	Διεργασία Β
`. . .`	`. . .`
`P(S)`	`P(S)`
`load R1,X ;κρίσιμο τμήμα Α`	`load R1,X ;κρίσιμο τμήμα Β`
`R1 = R1+1 ;κρίσιμο τμήμα Α`	`R1 = R1+1 ;κρίσιμο τμήμα Β`
`store X,R1 ;κρίσιμο τμήμα Α`	`store X,R1 ;κρίσιμο τμήμα Β`
`V(S)`	`V(S)`
`. . .`	`. . .`

Μια από τις δύο διεργασίες θα κλειδώσει πρώτη το κρίσιμο τμήμα εκτελώντας `P(S)` οπότε η άλλη θα περιμένει. Μόλις η πρώτη διεργασία εκτελέσει `V(S)` τότε θα απελευθερωθεί η πρόσβαση στο κρίσιμο τμήμα και έτσι και η δεύτερη διεργασία μπορεί να εκτελέσει το δικό της. (Ας σημειωθεί ότι τα κρίσιμα τμήματα δεν είναι απαραίτητο να περιέχουν τον ίδιο κώδικα όπως στο παράδειγμα αυτό. Το κρίσιμο τμήμα ορίζεται από τη χρήση μιας κοινής μεταβλητής, εδώ του Χ, και όχι από τον κώδικα που βρίσκεται μέσα σ' αυτό). Με την προσθήκη των σηματοφορέων το πρόγραμμά μας έχει πλέον προκαθορισμένο αποτέλεσμα: μετά την εκτέλεση και των δύο διεργασιών η τιμή του Χ θα είναι 2.

Εκτός από τους δυαδικούς σηματοφορείς υπάρχουν και οι πολλαπλοί σηματοφορείς οι οποίοι είναι ακέραιοι αριθμοί και επιτρέπουν περιορισμένο αριθμό διεργασιών να βρίσκονται ταυτόχρονα στο κρίσιμο τμήμα τους. Στην περίπτωση αυτή οι λειτουργίες Ρ και V ορίζονται ως εξής:

P(S) /*κλείδωμα*/:

```
while(S==0)
   /*wait and do nothing*/;
S=S-1;        /*αν S>0 μπες μέσα στο ΚΤ και μείωσε το S κατά 1*/
```

V(S) /*ξεκλείδωμα*/:

```
S=S+1;
```

Αρχικά ο σηματοφορέας παίρνει την τιμή S=N όπου N είναι ο μέγιστος αριθμός των διεργασιών που μπορούν να βρίσκονται ταυτόχρονα στο κρίσιμο τμήμα.

6.1.2.1 Πρακτικοί τρόποι κλειδώματος της μνήμης

- **Κλείδωμα και στριφογύρισμα (spin-lock).**

Με τη μέθοδο αυτή μια διεργασία εξετάζει το περιεχόμενο μιας διεύθυνσης της μνήμης Δ (π.χ. Δ=210) και αυτό είναι 0 τότε περιμένει μέχρι η τιμή αυτή να γίνει 1 και μόνο τότε η διεργασία συνεχίζει την εκτέλεσή της. Αυτό προϋποθέτει ότι η διεργασία αυτή δεν κάνει τίποτ' άλλο καθ' όσο διάστημα η διεύθυνση Δ περιέχει την τιμή 0, παρά μόνο διαβάζει συνεχώς τη Δ για να δει πότε ακριβώς το περιεχόμενό της θα γίνει μηδέν. Ο παρακάτω κώδικας περιγράφει τη μέθοδο αυτή:

```
      mov   R2,#0       ;R2 = 0 = τιμή κλειδώματος
wait: aexch R2,(210)    ;ατομική ανταλλαγή μεταξύ
                        ;καταχωρητή R2 και διεύθυνσης 210
      beqz  R2,wait     ;αν η τιμή που διαβάζεται από το 210
                        ;είναι 0 τότε έχει
                        ;κλειδωθεί από κάποια άλλη διεργασία
                        ;οπότε γύρνα πίσω και ξαναπροσπάθησε
      ....              ;υπόλοιπος κώδικας
```

Σχήμα 35. Κλείδωμα με χρήση ατομικής ανταλλαγής. Χρησιμοποιείται η μέθοδος της αναμονής με στριφογύρισμα (spin-lock) γύρω από τη θέση μνήμης 210. Ο κώδικας αυτός υλοποιεί ουσιαστικά η λειτουργία P για το σηματοφορέα που βρίσκεται στη διεύθυνση 210.

```
      mov   R2,#1       ;R2 = 1 = τιμή ξεκλειδώματος
      aexch R2,(210)    ;ατομική ανταλλαγή: [210] = 1
      ....              ;υπόλοιπος κώδικας
```

Σχήμα 36. Ξεκλείδωμα με χρήση ατομικής ανταλλαγής. Υλοποιεί ουσιαστικά τη λειτουργία V για το σηματοφορέα που βρίσκεται στη διεύθυνση 210.

Ο κώδικας αναμονής με στριφογύρισμα (spin) 4.1 χρησιμοποιείται από μια διεργασία η οποία περιμένει μέχρι η διεύθυνση 210 να πάρει την τιμή 1. Όπως φαίνεται και πιο πάνω η διεργασία μπαίνει σε ένα συνεχές loop το οποίο τερματίζεται μόνο αν [210] = 1. Λέμε ότι η διεργασία *στριφογυρίζει* (κάνει *spin*) γύρω από την κλειδαριά καθ' όσο διάστημα αυτή διαβάζει τη θέση μνήμης συνεχώς μέχρι να δει 1. Από κει και το όνομα spin-lock. Ο κώδικας ξεκλειδώματος 4.2 χρησιμοποιείται από μια διεργασία που εξέρχεται από το κρίσιμο τμήμα της.

Αυτή είναι και η πιο απλή μέθοδος κλειδώματος αλλά βέβαια έχει το μειονέκτημα ότι απαιτεί συνεχή απασχόληση της CPU και της μνήμης με συνεχή ανάγνωση από την ίδια διεύθυνση. Αυτό φορτώνει πολύ το δίαυλο της μνήμης (memory bus), όπως μπορεί εύκολα να γίνει κατανοητό, και δεσμεύει τον επεξεργαστή ο οποίος περιστρέφεται σε αναμονή.

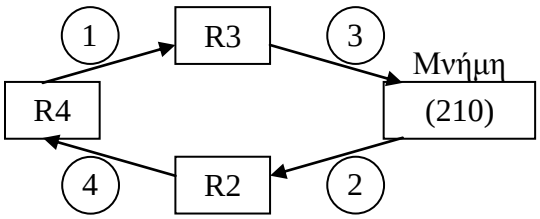
- **Οι εντολές load-linked, και store-conditional.**

Μια εναλλακτική λύση είναι η χρήση δύο εντολών αντί μιας αδιάσπαστης εντολής. Οι εντολές αυτές είναι οι `ll` (load-linked) και `sc` (store-conditional). Θα δούμε τι ακριβώς κάνουν οι εντολές αυτές και πώς χρησιμοποιούνται σε ένα παράδειγμα. Έστω ότι θέλουμε να κλειδώσουμε τη θέση μνήμης 210 με την τιμή 1 και έστω ότι η τιμή 1 βρίσκεται στον καταχωρητή R4. Αντί να κάνουμε `aexch R4,(210)`, εκτελούμε τον παρακάτω κώδικα

```
     ld   R4,#0          ;R4 = 0
try: mov  R3,R4          ;φόρτωσε προσωρινά τον R4 στον R3
     ll   R2,(210)       ;load-linked
     sc   R3,(210)       ;store-conditional
     beqz R3,try         ;άλμα πίσω αν το store απέτυχε (αν R3=0)
     mov  R4,R2          ;μεταφορά του παλαιού περιεχομένου
                         ;της διεύθυνσης 210 στον R4
```

Σχήμα 37. Κλείδωμα με χρήση των εντολών load-linked και store-conditional.

Το παρακάτω σχήμα δείχνει ακριβώς τα βήματα που γίνονται με τη σειρά που γίνονται



- Στο βήμα 1 η εντολή `mov R3,R4` μεταφέρει τα περιεχόμενα του καταχωρητή R4 στον R3.
- Στο βήμα 2 η εντολή `ll R2,(210)` μεταφέρει το περιεχόμενο της διεύθυνσης μνήμης 210 στον καταχωρητή R2. Η διαφορά της εντολής `ll` από την απλή `load` είναι η εξής: στην `ll` χρησιμοποιείται ένας καταχωρητής που λέγεται link register και ο οποίος φυλάει την διεύθυνση μνήμης που εμφανίζεται σαν δεύτερο όρισμα της `ll` (στη συγκεκριμένη περίπτωση το 210). Επί πλέον ο link register δεν είναι ένας απλός καταχωρητής. Έχει σχεδιαστεί έτσι ώστε αν γίνει οποιαδήποτε εγγραφή στη διεύθυνση 210 ο καταχωρητής αυτόματα μηδενίζεται. Η χρησιμότητα αυτού του γεγονότος θα φανεί αμέσως παρακάτω.
- Στο βήμα 3 σώζεται ο καταχωρητής R3 στη διεύθυνση μνήμης 210 με την εντολή `sc R3,(210)`. Η εντολή `sc` (store conditional) διαφέρει από την απλή `store` στο γεγονός ότι χρησιμοποιεί τον καταχωρητή link register. Αν η διεύθυνση που εμφανίζεται σαν δεύτερο όρισμα της sc (εδώ το 210) είναι ίδια με το περιεχόμενο του link register τότε το store εκτελείται κανονικά. Αυτό σημαίνει ότι καμία άλλη διεργασία δεν πείραξε το περιεχόμενο της διεύθυνσης 210 εν τω μεταξύ (δηλαδή ανάμεσα στις εντολές `ll` και `sc`). Αν όμως το δεύτερο όρισμα της `sc` διαφέρει από το περιεχόμενο του link register αυτό σημαίνει ότι ο link register έχει μηδενιστεί γιατί κάποια άλλη διεργασία πείραξε τη διεύθυνση 210. Στην περίπτωση αυτή το `sc` αποτυγχάνει και επιστρέφει την τιμή 0 (δηλαδή στην περίπτωση αυτή θα έχουμε R3 ← 0). Το άλμα υπό συνθήκη `beqz R3,try` κοιτάει την τιμή που επέστρεψε η `sc` στο R3 και αν είναι μηδέν (δηλαδή αποτυχία) ξαναπροσπαθεί πάλι από την αρχή.
- Στο βήμα 4 αν η τιμή του `R3` δεν είναι 0 (δηλαδή έχουμε επιτυχία του `sc`) τότε μεταφέρεται το περιεχόμενο του R2 στον R4. Το συνολικό αποτέλεσμα είναι μια ατομική ανταλλαγή του R4 με τη διεύθυνση μνήμης 210.

Το πλεονέκτημα της μεθόδου αυτής είναι η απλότητα σχεδιασμού του υλικού αφού δεν απαιτείται η ταυτόχρονη ανάγνωση και εγγραφή της μνήμης μέσα σε ένα κύκλο. Χρειάζεται βέβαια η σχεδίαση του link register ο οποίος θα πρέπει να παρακολουθεί την κίνηση στο bus της μνήμης και να μηδενίζεται κατάλληλα όταν εμφανίζεται η αντίστοιχη διεύθυνση στις γραμμές διεύθυνσης (address lines). Ωστόσο η σχεδίαση του link register δεν είναι τόσο απαιτητική και δύσκολη όσο η σχεδίαση μιας μνήμης που θα μπορεί να διαβάζει και να γράφει ταυτόχρονα μέσα σε ένα κύκλο.

Ένα ακόμα πλεονέκτημα του ζεύγους των εντολών `ll/sc` είναι ότι μπορεί κανείς να τις χρησιμοποιήσει για να υλοποιήσει και άλλες λειτουργίες συγχρονισμού όπως την fetch-and-increment που δείχνεται παρακάτω

```
try: ll R2,(210)        ;load-linked
     add R2,#1          ;R2 ← R2+1
     sc R2,(210)        ;store conditional
     beqz R2,try        ;άλμα πίσω αν το store απέτυχε
```

Σχήμα 38. Κώδικας fetch-and-increment με χρήση load-linked/store-conditional.

6.1.3 Συγχρονισμός διεργασιών

Η βασική λειτουργία του λογισμικού εδώ είναι το *φράγμα συγχρονισμού* (*synchronization barrier*). Ένα φράγμα αναγκάζει όλες τις διεργασίες να περιμένουν μέχρι να φτάσουν όλες στο φράγμα οπότε και απελευθερώνονται πάλι όλες ταυτόχρονα. Η τυπική υλοποίηση ενός φράγματος γίνεται με δύο κλειδαριές στριφογυρίσματος (spin-locks): η μια κλειδαριά προστατεύει τον κοινό μετρητή που μετράει πόσες διεργασίες έχουν ήδη φτάσει στο φράγμα και η άλλη κρατάει τις διεργασίες σε αναμονή μέχρι να φτάσει και η τελευταία διεργασία στο φράγμα. Έστω ότι οι λειτουργίες lock (κλείδωμα), unlock (ξεκλείδωμα), και spin (στριφογύρισμα) έχουν υλοποιηθεί από το λειτουργικό σύστημα, π.χ. με τη χρήση της ατομικής ανταλλαγής ή με τη χρήση των εντολών load-link και store-conditional που περιγράψαμε παραπάνω. Η περιγραφή ενός φράγματος συγχρονισμού σε μια γλώσσα υψηλού επιπέδου (όπως η C) φαίνεται στο παρακάτω κομμάτι κώδικα:

```
lock(κλειδί-του-count);      /* προστάτεψε την κοινή μεταβλητή count */
if (count==0) release=0;     /* αρχικοποίηση του release */
count = count+1;             /* μέτρα πόσοι έφτασαν στο φράγμα */
unlock(κλειδί-του-count);    /* τέλος κρίσιμου τμήματος για το count */
if (count==total) {          /* έφτασαν όλοι στο φράγμα */
    count=0;                 /* μηδένισε το μετρητή */
    release=1;               /* απελευθέρωσε τις διεργασίες */
}
else {
    spin(release);           /* περίμενε μέχρι να φτάσουν όλοι */
                                    }
```

Σχήμα 39. Υλοποίηση ενός απλού φράγματος συγχρονισμού σε γλώσσα υψηλού επιπέδου όπως η C.

Ο κώδικας του Σχ. 39 κλειδώνει την κοινή μεταβλητή count κάθε φορά που αυτή πρέπει να αυξηθεί. Την πρώτη φορά που μια διαδικασία μπαίνει στο φράγμα το flag `release` γίνεται 0. Όταν ο μετρητής φτάσει στην τιμή `total`, δηλαδή όταν όλες οι διαδικασίες

έχουν φτάσει στο φράγμα, τότε ο μετρητής μηδενίζεται και γίνεται `release=1`. Όλες οι διαδικασίες που περίμεναν στριφογυρίζοντας γύρω από το `release` απελευθερώνονται. Η εντολή `spin(release)` διαβάζει την μεταβλητή `release` μέχρι αυτή να γίνει 1 οπότε και εξέρχεται.

Ο απλός κώδικας του Σχήματος 39 έχει ωστόσο ένα βασικό μειονέκτημα. Συχνά το φράγμα βρίσκεται μέσα σε ένα loop ώστε μια διαδικασία που απελευθερώνεται από το φράγμα θα κάνει κάποια δουλειά και θα ξαναγυρίσει σε αυτό. Έστω τώρα ότι κάποια διεργασία δεν εγκαταλείπει το φράγμα παρ' όλο που `release=1`. Αυτό π.χ. θα μπορούσε να συμβεί αν το λειτουργικό σύστημα έχει δώσει μικρή προτεραιότητα σ' αυτή τη διεργασία. Έστω επίσης ότι κάποια άλλη διεργασία που απελευθερώθηκε είναι τόσο γρήγορη ώστε ξαναγυρίζει στο φράγμα πριν φύγει από αυτό η τελευταία διεργασία. Τώρα η αργή διεργασία έχει παγιδευτεί διότι η γρήγορη έχει κάνει `release=0`. Ακόμη χειρότερα, όλες οι άλλες διεργασίες θα παγιδευτούν την μόλις φτάσουν το φράγμα διότι ο μετρητής δε θα φτάσει ποτέ την τιμή `total`.

Το πρόβλημα λύνεται με μια απλή τροποποίηση του κώδικα (Σχ. 40) χρησιμοποιώντας μια τοπική μεταβλητή `local_sense` για κάθε διεργασία:

```
local_sense = 1 - local_sense;
lock(κλειδί-του-count);       /* προστάτεψε την κοινή μεταβλητή count */
  count = count+1;            /* μέτρα πόσοι έφτασαν στο φράγμα */
  unlock(κλειδί-του-count);   /* τέλος κρίσιμου τμήματος για το count */
  if (count==total) {         /* έφτασαν όλοι στο φράγμα */
      count=0;                /* μηδένισε το μετρητή */
      release=local_sense;    /* απελευθέρωσε τις διεργασίες */
  }
  else {
      spin(release=local_sense);    /* περίμενε μέχρι να φτάσουν όλοι */
                                  }
```

Σχήμα 40. Υλοποίηση ενός φράγματος συγχρονισμού με χρήση τοπικών μεταβλητών για την αποφυγή αδιεξόδων.

6.2 Αδιέξοδα και αποφυγή τους

Ένα από τα σημαντικότερα προβλήματα του συγχρονισμού μεταξύ των διαφόρων διεργασιών που εκτελούνται παράλληλα είναι και η αποφυγή καταστάσεων όπου καμία διεργασία δεν προχωρεί διότι όλες οι εργασίες κρατάνε κάποια αγαθά του συστήματος εμποδίζοντας η μια την άλλη να συνεχίσουν την εκτέλεσή τους. Οι καταστάσεις αυτές καλούνται *αδιέξοδα* και οδηγούν σε άπειρη αναμονή όλων των διεργασιών δηλαδή σε ολοκληρωτικό σταμάτημα της λειτουργίας του συστήματος.

Αδιέξοδα δημιουργούνται όταν ισχύουν ταυτόχρονα οι τέσσερεις παρακάτω συνθήκες:

1. *Αμοιβαίος αποκλεισμός (mutual exclusion)*. Κάθε διεργασία έχει τον αποκλειστικό έλεγχο των αγαθών του συστήματος που της έχουν διατεθεί.

2. *Όχι προ-απελευθέρωση (no pre-emption)*. Μια διεργασία δεν απελευθερώνει τα αγαθά που διαθέτει παρά μόνο αφού τελειώσει τη χρήση τους.

3. *Δέσμευση και αναμονή (hold-and-wait)*. Μια διεργασία επιτρέπεται να δεσμεύσει κάποια αγαθά και ταυτόχρονα να περιμένει για να της διατεθούν και άλλα αγαθά.

4. *Κυκλική αναμονή (circular wait)*. Πολλές διαφορετικές διεργασίες περιμένουν η μια την άλλη να απελευθερώσει τα αγαθά της με κυκλικό τρόπο. Δηλαδή η διεργασία Δ_1 περιμένει την Δ_2 η οποία περιμένει την Δ_3 η οποία ... η οποία περιμένει την Δ_N η οποία περιμένει την Δ_1.

Η αποφυγή αδιεξόδων είναι ένα από τα βασικά μελήματα ενός παράλληλου λειτουργικού συστήματος και γενικά ενός παράλληλου προγράμματος.

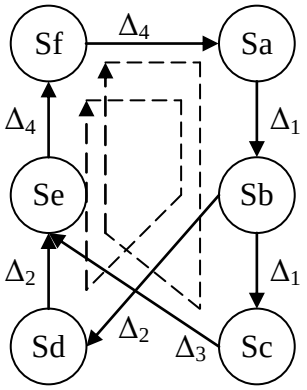
Παράδειγμα: Στατική αποφυγή αδιεξόδων.

Θεωρήστε τις 4 παρακάτω διεργασίες οι οποίες μοιράζονται τους σηματοφορείς Sa, Sb, ..., Sf και υποθέστε ότι κάθε ένας από αυτούς τους σηματοφορείς επιτρέπει μόνο σε μια διεργασία να τον δεσμεύσει. Ο σηματοφορέας Sx δεσμεύεται με την κλήση του τελεστή P(Sx) και αποδεσμεύεται με την κλήση του V(Sx). Στις διεργασίες αυτές έχουμε παραλείψει να δείξουμε τον κύριο κώδικα ο οποίος δεν μας ενδιαφέρει αλλά δείχνουμε μόνο τη σειρά των τελεστών P, V για κάθε διεργασία.

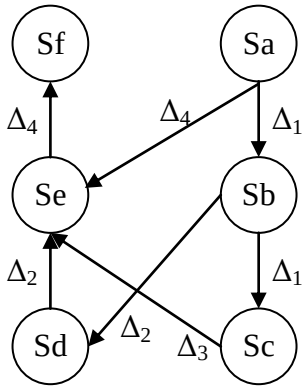
Διεργασία Δ1	**Διεργασία Δ2**	**Διεργασία Δ3**	**Διεργασία Δ4**	**Διεργασία Δ4 (τροποποιημένη)**
...	P(Sb)	...	...	...
P(Sa)	...	P(Sc)	...	...
...	P(Sd)	...	P(Se)	P(Sa)
P(Sb)	...	P(Se)	...	P(Se)
...	P(Se)	...	P(Sf)	...
P(Sc)	...	...	...	P(Sf)
...	...	...	P(Sa)	...
...	V(Se)	V(Se)	...	...
...	...	...	...	...
V(Sa)	...	...	V(Sf)	V(Sf)
...	V(Sd)	...	V(Sa)	V(Sa)
V(Sb)	...	V(Sc)	...	...
V(Sc)	V(Sb)	...	V(Se)	V(Se)
...	...	...	...	...

Αν οι διεργασίες εκτελούνται ασύγχρονα τότε δεν μπορούμε να ξέρουμε με ποια σχετική σειρά θα εκτελεστούν οι εντολές μεταξύ διαφορετικών διεργασιών. Το μόνο που γνωρίζουμε είναι η σχετική σειρά εκτέλεσης των εντολών μέσα στην ίδια διεργασία. Αν οι διεργασίες αιτήσουν τους σηματοφορείς με οποιαδήποτε σειρά που να δημιουργεί κύκλο τότε έχουμε αδιέξοδο.

Το πρόβλημα μπορεί να περιγραφεί πολύ περιεκτικά με τη χρήση ενός γράφου όπως ο κάτω αριστερά [γράφος (α)]:



(α) Γράφος δέσμευσης αγαθών που αντιστοιχεί στις διεργασίες Δ1, Δ2, Δ3, Δ4. Ο γράφος αυτός δείχνει την ύπαρξη δύο κύκλων.



(β) Μια απλή μετατροπή στη σειρά δέσμευσης των σηματοφορέων στη διεργασία Δ4 λύνει το πρόβλημα των κύκλων.

Σχήμα 41. Αποφυγή αδιεξόδων χρησιμοποιώντας γράφους δέσμευσης αγαθών.

Κάθε κόμβος στο γράφο αντιστοιχεί σε ένα κοινό αγαθό (στη συγκεκριμένη περίπτωση σε ένα σηματοφορέα). Κάθε ακμή στο γράφο δείχνει ότι μια διεργασία δεσμεύει ένα σηματοφορέα και ταυτόχρονα αιτεί τη δέσμευση και ενός άλλου. Έτσι η ακμή μεταξύ των κόμβων Sa και Sb με πινακίδα Δ1 μας λέει ότι η διεργασία Δ1 πρώτα δεσμεύει τον σηματοφορέα Sa και κατόπιν αιτεί τη δέσμευση του Sb. Αυτό σημαίνει ότι στον κώδικα της Δ1 υπάρχει η εντολή P(Sa) που ακολουθείται από την P(Sb).

Ο γράφος (α) μας δείχνει ότι υπάρχει δυνατότητα να συμβεί αδιέξοδο αφού υπάρχουν δύο διαφορετικοί κύκλοι μέσα σ' αυτόν (ο κύκλος Sa-Sb-Sc-Se-Sf και ο κύκλος Sa-Sb-Sd-Se-Sf). Οι κύκλοι αυτοί φαίνονται με διακεκομμένες γραμμές στο σχήμα και μας λένε ακριβώς με ποια ακολουθία εντολών θα συμβεί το αδιέξοδο. Έτσι για να συμβεί ο κύκλος Sa-Sb-Sc-Se-Sf πρέπει η διεργασία Δ1 να δεσμεύσει τους σηματοφορείς Sa και Sb, η Δ3 να δεσμεύσει τον Sc, και η Δ4 να δεσμεύσει τους Se, Sf. Αν συμβεί αυτό τότε η Δ1 θα περιμένει τον σηματοφορέα Sc που είναι δεσμευμένος από την Δ3, η Δ3 θα περιμένει τον Se που είναι δεσμευμένος από την Δ4, και τέλος η Δ4 θα περιμένει τον Sa που είναι δεσμευμένος από την Δ1. Αποτέλεσμα φυσικά θα είναι το αδιέξοδο.

Μια απλή λύση του προβλήματος είναι η αντιστροφή ή η κατάργηση κάποιων ακμών στον γράφο (α) και η δημιουργία του γράφου (β) στον οποίο και οι δύο παραπάνω κύκλοι έχουν εξαφανιστεί, και ούτε υπάρχουν άλλοι κύκλοι. Πώς υλοποιείται η αλλαγή της ακμής προγραμματιστικά; Απλούστατα, στην διεργασία Δ4 μετακινούμε την εντολή P(Sa) πριν από την εντολή P(Se) και την P(Sf). Έτσι καταργείται η ακμή Sf-Sa, και δημιουργείται η ακμή Sa-Se, καταστρέφοντας και τους δύο κύκλους.

Παρατηρήστε ότι τέτοιες πολύ απλές αλλαγές είναι πολύ αποτελεσματικές αλλά και εύκολες να εντοπιστούν μηχανικά από ένα βελτιστοποιημένο μεταφραστή (compiler). Παρατηρήστε επίσης ότι σε όλη την παραπάνω συζήτηση δε λαμβάνουμε καθόλου υπ' όψη μας τη σειρά των εντολών V( ). Οι εντολές V( ) μας ενδιαφέρουν μόνο εφόσον παρεμβάλλονται μεταξύ δύο εντολών P και αφορούν τον πρώτο σηματοφορέα. Για παράδειγμα, ο γράφος που αντιστοιχεί στον κώδικα

P(S1)
V(S1)
P(S2)

δεν περιέχει ακμή μεταξύ των κόμβων S1 και S2 διότι η διεργασία έχει πρώτα απελευθερώσει τον S1 πριν αιτήσει τον S2. Αντίθετα ο γράφος του κώδικα

P(S1)
P(S2)
V(S1)

έχει ακμή μεταξύ των κόμβων S1 και S2 διότι η διεργασία κρατάει δεσμευμένο τον S1 και κατόπιν ζητάει τον S2 χωρίς να έχει πρώτα αποδεσμεύσει τον S1.

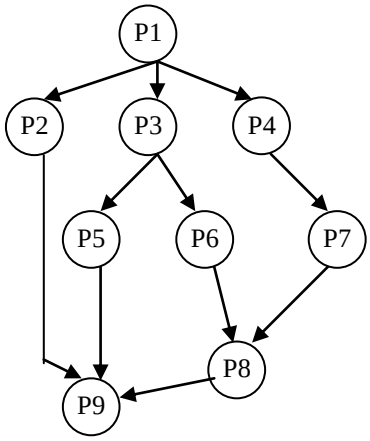
6.3 Παράλληλες εντολές και δομές υψηλού επιπέδου

6.3.1 Οι εντολές fork και join

Η εντολή `fork L` δημιουργεί ένα αντίγραφο της παρούσης διεργασίας το οποίο ξεκινάει να εκτελείται όχι από την αρχή αλλά από το σημείο `L`. Το `L` δεν είναι τίποτ' άλλο παρά μια πινακίδα (label) μέσα στο πρόγραμμα.

Αντίστοιχα η εντολή `join n` είναι ουσιαστικά ένα φράγμα συγχρονισμού για `n` διεργασίες. Μόλις μια διεργασία φτάσει στην εντολή `join n` περιμένει να φτάσουν και οι άλλες διεργασίες του προγράμματος που συναντούνται στο σημείο αυτό και αφού φτάσουν όλες εκεί τότε καταστρέφονται όλες πλην μιας η οποία συνεχίζει την εκτέλεσή της.

Οι εντολές αυτές μπορούν να γίνουν καλύτερα κατανοητές με ένα παράδειγμα. Έστω ότι μας δίνεται ο παρακάτω αλγόριθμος υπό μορφή γράφου προήγησης:



Σχήμα 42. Γράφος προήγησης ενός υποθετικού αλγορίθμου.

Οι διαδικασίες P1, P2, ..., P9 είναι κάποιες υπορουτίνες που εκτελούν κάποιες συγκεκριμένες υπολειτουργίες του αλγορίθμου. Ο γράφος προήγησης δείχνει την εξάρτηση μεταξύ των υπορουτινών αυτών. Για παράδειγμα η υπορουτίνα P4 εξαρτάται από την P1 διότι προφανώς παίρνει σαν εισόδους τα αποτελέσματα της P1. Λογικά η εκτέλεση της P4 πρέπει να ακολουθεί χρονικά την εκτέλεση της P1. Η παράλληλη υλοποίηση του αλγορίθμου 4.8 με χρήση των εντολών fork και join φαίνεται στο παρακάτω σχήμα:

```
     count8 = 2;  /*μέτρα πόσες υποροutίνες συγκλίνουν πριν την P8*/
     count9 = 3;  /*μέτρα πόσες υποροutίνες συγκλίνουν πριν την P9*/
L1:  P1; fork L2; fork L3; goto L4;
L2:  P2; goto L9;
L3:  P3; fork L5; goto L6;
L4:  P4; goto L7;
L5:  P5; goto L9;
L6:  P6; goto L8;
L7:  P7; goto L8;
L8:  join count8;
     P8;
L9:  join count9;
     P9;
```

Σχήμα 43. Παράλληλος κώδικας με fork/join για την υλοποίηση του αλγορίθμου του Σχήματος 42.

Ο κώδικας του Σχ. 43 μπορεί να παραχθεί αυτόματα από το σχήμα 42 αν ακολουθηθούν τα παρακάτω βήματα.

- Βήμα 1. Διαλέγουμε μια ξεχωριστή πινακίδα (label) για κάθε υπορουτίνα.

- Βήμα 2. Ανάλογα με τα παιδιά κάθε υπορουτίνας στο δέντρο γράφουμε τις αντίστοιχες εντολές `fork` kai `goto` μετά από την υπορουτίνα. Συγκεκριμένα:
  - Αν μια υπορουτίνα P έχει ένα μόνο παιδί με πινακίδα L1 τότε γράφουμε
    `P; goto L1;`
  - Αν μια υπορουτίνα P έχει δύο παιδιά με πινακίδες L1, L2, τότε γράφουμε

```
P; fork L1; goto L2;
```

- Αν μια υπορουτίνα P έχει τρια παιδιά με πινακίδες L1, L2, L3, τότε γράφουμε
```
P; fork L1; fork L2; goto L3;
```
- κλπ...

➢ Βήμα 3. Ανάλογα με τον αριθμό των ακμών που εισέρχονται σε ένα κόμβο του γράφου προήγησης γράφουμε την αντίστοιχη εντολή join πριν από την υπορουτίνα.
- Αν σε μια υπορουτίνα P εισέρχεται μόνο μια ακμή τότε δε γράφουμε τίποτα πριν από το `P;`
- Αν σε μια υπορουτίνα P εισέρχονται n ακμές (όπου n>1) τότε γράφουμε
```
join count;
```
πριν από το `P;`. Έχουμε φροντίσει να αρχικοποιήσουμε το `count` στην τιμή n στην αρχή του προγράμματος.

Όπως είναι προφανές, με τις εντολές fork και join μπορούμε γενικά να υλοποιήσουμε οποιονδήποτε γράφο προήγησης.

6.3.2 Οι εντολές parbegin, parend

Οι εντολές parbegin (*par*allel *begin*) και parend (*par*allel *end*) ορίζουν ένα block κώδικα όπου οι διάφορες εντολές εκτελούνται παράλληλα. Για παράδειγμα ο παρακάτω κώδικας

```
parbegin
   P1 || P2 || P3
parend;
```

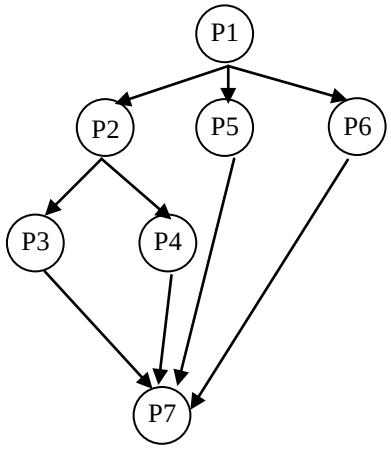
εκτελεί παράλληλα τις υπορουτίνες P1, P2, P3. Το parend εκτελείται μόνο αφού όλες οι υπορουτίνες P1, P2, P3 έχουν ολοκληρωθεί. Με άλλα λόγια η εντολή parend είναι ένα φράγμα συγχρονισμού. Το σύμβολο `||` διαχωρίζει τα τμήματα του κώδικα που εκτελούνται παράλληλα.

Είναι δυνατόν να υπάρχουν πολλές εντολές parbegin-parend φωλιασμένες η μια μέσα στην άλλη όπως και τα for-loops. Για παράδειγμα, ο παρακάτω κώδικας

```
P1;
parbegin
   {
   P2;
   parbegin
      P3 || P4
   parend
   }
   || P5 || P6
parend;
P7;
```

Σχήμα 44. Παράδειγμα κώδικα με χρήση parbegin-parend.

εκτελεί τον ακόλουθο γράφο προήγησης

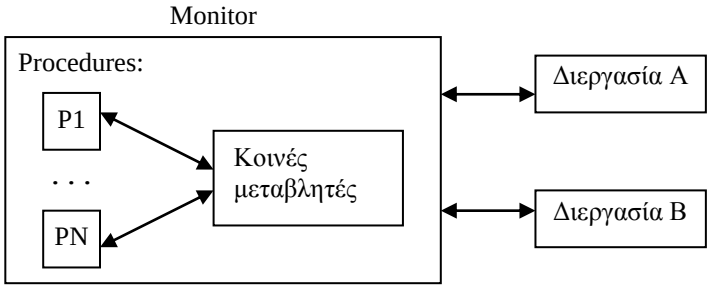


Σχήμα 45. Ο Γράφος προήγησης που αντιστοιχεί στον κώδικα του Σχ. 44.

Οι εντολές parbegin και parend δεν είναι τόσο ισχυρές όσο οι fork/join και δεν μπορούν να περιγράψουν εν γένει έναν οποιοδήποτε γράφο προήγησης παρά μόνο αυτούς που έχουν ειδική μορφή.

6.3.3 Η δομή monitor

Το monitor είναι μια δομή υψηλού επιπέδου η οποία κανονικά υλοποιείται από το λειτουργικό σύστημα και υποστηρίζει τον δομημένο προγραμματισμό. Όπως φαίνεται και στο παρακάτω σχεδιάγραμμα το monitor ελέγχει την πρόσβαση σε κοινές μεταβλητές και όλες οι διεργασίες που ζητούν τη χρήση τους πρέπει απαραίτητα να χρησιμοποιήσουν το monitor. Έτσι υπάρχει πλήρης κεντρικός έλεγχος στα κοινά αγαθά και γίνεται σωστότερη διαχείρισή τους.




```
Construct:
Monitor secretary(name)
/* declaration of local data */
Procedure P1(parameter list)
   begin
   ...
   end
Procedure P2(parameter list)
```

```
   begin
   ...
   end

......

Procedure PN(parameter list)
   begin
   ...
   end
begin
   /* initialization of local data */
end
```

Μια δομή monitor αποτελείται γενικά από τρία μέρη. Στο πρώτο μέρος ορίζεται το όνομα του monitor και οι τοπικές μεταβλητές που χρησιμοποιούνται σ' αυτό. Το δεύτερο μέρος είναι μια συλλογή από procedures οι οποίες χρησιμοποιούν τις δηλωθείσες μεταβλητές. Τέλος στο τρίτο μέρος γίνεται αρχικοποίηση όλων των μεταβλητών.

Η χρήση του monitor γίνεται στον προγραμματισμό υψηλού επιπέδου όπου τα κρίσιμα τμήματα όλων των διεργασιών αφαιρούνται από τις διεργασίες και προστίθενται σαν procedures μέσα στο monitor. Έτσι για παράδειγμα, το κρίσιμο τμήμα μιας διεργασίας Α που περιέχει π.χ. μια κοινή μεταβλητή Χ, μεταφέρεται σαν υπορουτίνα Ρ1 στο monitor και η μεταβλητή Χ σαν τοπική μεταβλητή του monitor. Όταν η Α επιθυμήσει να εκτελέσει το κρίσιμο τμήμα της θα εκτελέσει την ρουτίνα Ρ1 του monitor.

Το monitor σαν δομή που υποστηρίζεται από το λειτουργικό σύστημα αναλαμβάνει την προστασία (το κλείδωμα και ξεκλείδωμα) των τοπικών του μεταβλητών για να μην δημιουργούνται προβλήματα από την κοινοκτημοσύνη και την παράλληλη πρόσβαση. Αυτό αποδεσμεύει τον προγραμματιστή από λάθη και από το βάρος της ευθύνης να προστατέψει σωστά το κρίσιμο τμήμα κάθε διεργασίας που γράφει. Αντί αυτού ο προγραμματιστής γράφει το κρίσιμο τμήμα κάθε διεργασίας σε ένα monitor και όποτε το χρειάζεται το καλεί μέσα από το monitor. Αυτό διευκολύνει πολύ το debugging και την ανάπτυξη του παράλληλου κώδικα. Φυσικά κάθε διεργασία μπορεί να ορίσει πάνω από ένα monitor ανάλογα με τις ανάγκες του προγράμματος.

Παράδειγμα: Το μοντέλο παραγωγού/καταναλωτή μέ χρήση της δομής monitor.

Στο μοντέλο παραγωγού-καταναλωτή υπάρχει μια κοινή περιοχή που καλείται buffer και περιέχει Ν θέσεις μνήμης. Στο παράδειγμα αυτό ο buffer περιέχει ακεραίους αριθμούς αλλά θα μπορούσε να περιέχει οποιοδήποτε άλλο τύπο μεταβλητής ακόμη και χαρακτήρες. Ο παραγωγός μοντελοποιείται από μια διαδικασία deposit(x) ενώ ο καταναλωτής από μια διεργασία fetch(x). Και οι δύο πρέπει να έχουν προσπέλαση στον κοινό χώρο μνήμης του buffer. Η μεν υπορουτίνα deposit γράφει έναν αριθμό στον buffer στη θέση inpointer και αυξάνει τον inpointer κατά 1 η δε υπορουτίνα fetch διαβάζει ένα αριθμό από τη θέση

outpointer του buffer και μειώνει τον outpointer κατά 1. Οι ρουτίνες αυτές φαίνονται στην παρακάτω δομή monitor όπου γίνεται και ο κατάλληλος έλεγχος για το overflow και το underflow του buffer:

```
Monitor Producer_Consumer

  /* Δήλωση τοπικών μεταβλητών */
  integer: buffer[0:N-1];
  integer: inpointer, outpointer;
  Boolean: notfull, notempty;
  integer: count;

  /* Δήλωση τοπικών procedures */
  Procedure deposit(x)
     begin
     if (count == N) wait(notfull);
     buffer(inpointer) = x;
     inpointer = (Inpointer + 1) mod N;
     signal(notempty);
     end

  Procedure fetch(x)
     begin
     if (count==0) wait(notempty);
     x = buffer(outpointer);
     outpointer = (outpointer + 1) mod N;
     signal(notfull);
     end

  /* Αρχικοποίηση τοπικών μεταβλητών */
  begin
inpointer = 0;
  outpointer = 0;
  count = 0;
  end
```

Κάθε διεργασία που είναι παραγωγός πρέπει υποχρεωτικά να καλέσει τη ρουτίνα deposit() του monitor και κάθε διεργασία καταναλωτής πρέπει υποχρεωτικά να καλέσει τη ρουτίνα fetch() του monitor. Το monitor επιτρέπει κάθε φορά μόνο σε μια διεργασία να χρησιμοποιήσει μια ρουτίνα. Επομένως αν υπάρχουν πολλοί παραγωγοί που θέλουν να χρησιμοποιήσουν την deposit θα μπουν σε μια σειρά και ο καθένας θα εκτελεί την ρουτίνα του αδιάσπαστα. Όταν θα τελειώνει αυτός τότε θα εκτελεί τη ρουτίνα ο επόμενος στη σειρά. Ομοίως φυσικά θα γίνεται και με τους καταναλωτές και την ρουτίνα fetch. Ένα κλασσικό παράδειγμα μοντέλου με πολλούς παραγωγούς-καταναλωτές είναι ένα δίκτυο επεξεργαστών όπου εκείνοι που θέλουν να στείλουν ένα μήνυμα είναι παραγωγοί και αυτοί που θέλουν να διαβάσουν ένα μήνυμα είναι οι καταναλωτές. Στην περίπτωση αυτή η δομή monitor διευκολύνει ιδιαίτερα την ανάπτυξη του λογισμικού για την παράλληλη επικοινωνία.