ΕΝΟΤΗΤΑ 3

ΣΥΝΤΡΕΧΩΝ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

Περιεχόμενα

1. Οι έννοιες του ταυτοχρονισμού και συντρέχοντα προγραμματισμού

- Με τον όρο *ταυτοχρονισμό* (concurrency) αναφερόμαστε στην ταυτόχρονη εκτέλεση δύο ή περισσοτέρων διεργασιών σε ένα σύστημα. Αυτό επιτυγχάνεται με τη συνεχή εναλλαγή των εκτελούμενων διεργασιών στην ΚΜΕ. Σημειωτέον ότι ο ταυτοχρονισμός διαφέρει από τον *παραλληλισμό* (parallelism) όπου το σύστημα διαθέτει πολλούς επεξεργαστές οι οποίοι εκτελούν παράλληλα περισσότερες της μίας διεργασίες.

- Το φαινόμενο του ταυτοχρονισμού δημιουργείται σε ένα Λ.Σ. λόγω της ύπαρξης πολλών ταυτόχρονα εκτελούμενων διεργασιών. Κατ' επέκταση δημιουργείται η ανάγκη παροχής στον προγραμματιστή μηχανισμών για την υποστήριξη τεχνικών προγραμματισμού βασισμένων στην έννοια του ταυτοχρονισμού όπως δημιουργία νέων διεργασιών, συντονισμός της ταυτόχρονης εκτέλεσης ενός αριθμού διεργασιών, *διαδιεργασιακή* (interprocess) *επικοινωνία*, δηλαδή επικοινωνία μεταξύ ταυτόχρονα εκτελούμενων διεργασιών, κλπ. Η μορφή αυτή προγραμματισμού αναφέρεται ως *συντρέχων* ή *σύνδρομος* (concurrent) προγραμματισμός.

- Ιστορικά, ο προπομπός του συντρέχοντος προγραμματισμού είναι ο μηχανισμός των *συρρουτινών* (coroutines) που προτάθηκε από τον Conway το 1963. Ο προγραμματισμός με συρρουτίνες ήταν προέκταση του προγραμματισμού με διαδικασίες (procedures): ενώ στην περίπτωση των διαδικασιών μία διαδικασία έπρεπε να αποπερατώσει την εκτέλεσή της πριν ο έλεγχος εκτέλεσης του προγράμματος επιστρέψει στη διαδικασία που την κάλεσε, στην περίπτωση των συρρουτινών ο έλεγχος εκτέλεσης του προγράμματος μπορούσε να μεταφερθεί από τη μία συρρουτίνα στην άλλη οποιαδήποτε στιγμή· επιπλέον, η καλούμενη συρρουτίνα επανέρχιζε την εκτέλεσή της από το σημείο που είχε σταματήσει.

- Η διαφορά της έννοιας των συρρουτινών από αυτή του συντρέχοντος προγραμματισμού είναι ότι μόνο μία συρρουτίνα είναι ανά πάσα χρονική στιγμή ενεργοποιημένη και ότι η εναλλαγή ενεργοποίησης των συρρουτινών γίνεται στατικά σε συγκεκριμένα σημεία ενός προγράμματος.

2. Συρρουτίνες

- Πρόβλημα: Να διαβαστούν κάρτες των 80 χαρακτήρων και να τυπωθούν σε γραμμές των 125 χαρακτήρων, μετά δε το τέλος κάθε κάρτας να εισαχθεί ένας κενός χαρακτήρας και κάθε δύο `**` να αντικαταστηθούν με ένα `#`.

- Σειριακή λύση με 3 διαδικασίες:

  Διαδικασία 1η: Διάβασε *όλες* τις κάρτες σε προσωρινή μνήμη (buffer).

  Διαδικασία 2η: Κάνε τις αντικαταστήσεις.

  Διαδικασία 3η: Τύπωσε το αποτέλεσμα σε γραμμές των 125 χαρακτήρων.

  Μειονεκτήματα: Χάσιμο χρόνου (overhead) λόγω του ότι εκτελούνται πολλές εντολές Ε/Ε αλλά και μεγάλη σπατάλη μνήμης.

- Λύση με συρρουτίνες:

```
var rs,sp: character;
inbuf: array[1..80] of character;
outbuf: array[1..125] of character;

procedure read;
begin
  repeat
    READCARD(inbuf);
    for i=1 to 80 do
    begin
      rs:=inbuf[i];
      RESUME squash;
    end;
    rs="";
    RESUME squash;
  forever
end;

procedure print;
begin
  repeat
    for j=1 to 125 do
    begin
      outbuf[j]:=sp;
      RESUME squash;
    end;
    OUTPUT(outbuf);
  forever
end;

procedure squash;
begin
  repeat
    if rs≠"*" then
    begin
          sp:=rs;
          RESUME print
    end
        else begin
             RESUME read;
              if rs="*" then
              begin
                sp:="#";
                RESUME print
              end
              else begin
                   sp:="*";
                  RESUME print;
                     sp:=rs;
                     RESUME print;
                end
              end
        RESUME read
  forever
end.
```

3. Αλληλεπίδραση μεταξύ διεργασιών

- Ο ταυτοχρονισμός δημιουργεί στο Λ.Σ. την ανάγκη αντιμετώπισης των ακόλουθων προβλημάτων:
  - — Διαχείρηση των εκτελούμενων διεργασιών.
  - — Κατανομή των διαθέσιμων πόρων αναφορικά με ΚΜΕ, κύρια και περιφερειακή μνήμη και συσκευές Ε/Ε μεταξύ των εκτελούμενων διεργασιών.
  - — Προστασία του περιβάλλοντος μίας διεργασίας από μη επιτρεπτές παρεμβολές από τις άλλες διεργασίες.
  - — Ανεξαρτησία των αποτελεσμάτων που παράγει μία διεργασία από την ταχύτητα εκτέλεσής της σε σχέση με τις άλλες διεργασίες του συστήματος.

- Η ταυτόχρονη εκτέλεση περισσοτέρων της μίας διεργασιών οδηγεί συχνά σε φαινόμενα *αλληλεπίδρασης* μεταξύ των διεργασιών αυτών (process interaction) που μπορούν να χωρισθούν σε 3 κατηγορίες ανάλογα με το βαθμό στον οποίο μία διεργασία γνωρίζει την ύπαρξη άλλων διεργασιών και πως τυχόν εκμεταλλεύεται αυτή την πληροφορία:

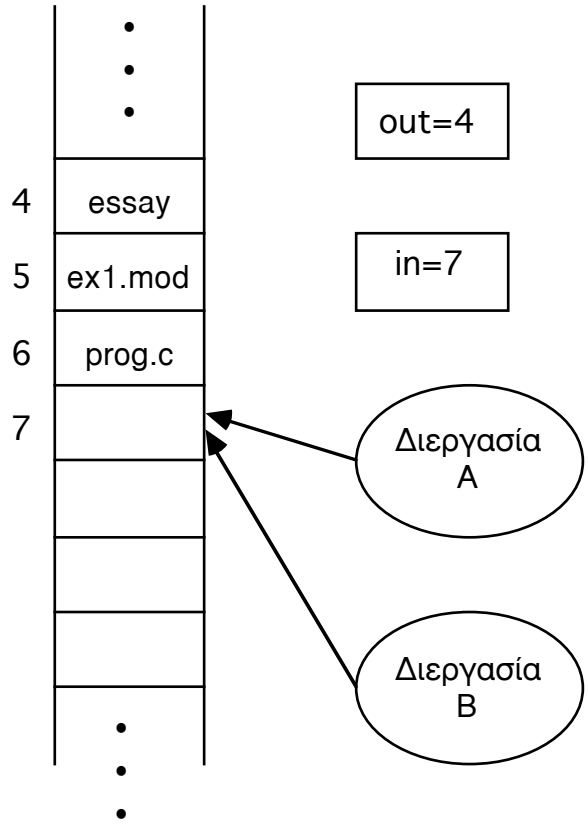
  Οι διεργασίες δεν γνωρίζουν η μία την ύπαρξη της άλλης (π.χ. οι διεργασίες που αντιστοιχούν στο κέλυφος ενός χρήστη στο Unix). Σε αυτή την περίπτωση δεν τίθεται θέμα υποστήριξης διαδιεργασιακής επικοινωνίας αλλά μπορεί να δημιουργηθεί πρόβλημα *ανταγωνισμού* (competition), δηλαδή προσπάθεια από τις εκτελούμενες διεργασίες να δεσμεύσουν τους ίδιους πόρους.

  Οι διεργασίες γνωρίζουν έμμεσα η μία την ύπαρξη της άλλης (π.χ. μέσω της κοινής χρήσης ενός διαμοιραζόμενου αντικειμένου όπως λ.χ. προσωρινή μνήμη (buffer)). Σε αυτή την περίπτωση δημιουργείται το φαινόμενο της *συνεργασίας* (cooperation) μέσω προσπέλασης στο κοινής χρήσης αντικείμενο.

  Οι διεργασίες γνωρίζουν άμεσα η μία την ύπαρξη της άλλης (π.χ. στη χρήση διοχετεύσεων στο Unix). Σε αυτή την περίπτωση οι διεργασίες γνωρίζουν η μία την άλλη κατ' όνομα και έχουν σχεδιασθεί για να εργάζονται μαζί για την επίτευξη ενός στόχου. Επομένως και εδώ έχουμε το φαινόμενο της συνεργασίας.

4. Ανταγωνισμός μεταξύ διεργασιών

- Δημιουργείται κυρίως όταν περισσότερες της μίας διεργασίες προσπαθούν ταυτόχρονα να δεσμεύσουν έναν πόρο. Κλασσικό παράδειγμα είναι αυτό του διεκπεραιωτή ετεροχρονισμένης εκτύπωσης (spooling):



- Η διεργασία Α διαβάζει την τιμή της μεταβλητής `in` που δείχνει στην επόμενη ελεύθερη θέση στον κατάλογο των αρχείων που περιμένουν να εκτυπωθούν, αλλά πριν ολοκληρώσει την εργασία της διακόπτεται η εκτέλεσή της και ξεκινά η εκτέλεση της διεργασίας Β, η οποία αφού διαβάσει και αυτή την τιμή της `in` αποθηκεύει στη θέση 7 το αρχείο προς εκτύπωση και ανεβάζει την τιμή της μεταβλητής `in` κατά μία μονάδα. Τώρα επαναρχίζει η εκτέλεση της διεργασίας Α από το σημείο διακοπής της, η οποία κάνοντας χρήση της παλιάς τιμής τής `in` τοποθετεί το δικό της αρχείο στη θέση 7 αντί της 8. Αποτέλεσμα είναι ότι το αρχείο της διεργασίας Β δεν θα εκτυπωθεί ποτέ. Όταν ένα τελικό αποτέλεσμα εξαρτάται από τη σειρά ή/και τον χρόνο εκτέλεσης δύο ή περισσοτέρων διεργασιών τότε δημιουργούνται *συνθήκες ανταγωνισμού* (race conditions).

4. Ανταγωνισμός μεταξύ διεργασιών (συνέχεια)

- Για την αποφυγή συνθηκών ανταγωνισμού επιβάλλεται ο *αμοιβαίος αποκλεισμός* (mutual exclusion), δηλαδή ο αποκλεισμός μίας διεργασίας από κάποια ενέργεια που ταυτόχρονα επιτελεί κάποια άλλη διεργασία. Αυτό οδηγεί στον καθορισμό κάποιων μερών στον κώδικα μίας διεργασίας στα οποία η διεργασία αυτή προσπαθεί να προσπελάσει κοινά μεταξύ των διεργασιών αντικείμενα (π.χ. διαμοιραζόμενες περιοχές μνήμης, αρχεία, κλπ.). Αυτά τα μέρη λέγονται *κρίσιμα τμήματα* (critical sections) και για τη σωστή εκτέλεση συντρέχουσων διεργασιών πρέπει να διασφαλιστεί ότι <u>ποτέ δύο ή περισσότερες διεργασίες δεν θα βρίσκονται ταυτόχρονα στο ίδιο κρίσιμο τμήμα</u>.

- Η εισαγωγή του αμοιβαίου αποκλεισμού δημιουργεί δύο ακόμα προβλήματα:
  - Του *αδιέξοδου* (deadlock) όταν λ.χ. δύο διεργασίες χρειάζονται δύο πόρους και κάθε μία από τις διεργασίες έχει δεσμεύσει ένα από τους πόρους και έχοντας αρχίσει οι διεργασίες να εκτελούν το κρίσιμο τμήμα του κώδικά τους για τον δεύτερο πόρο ανακαλύπτουν ότι είναι δεσμευμένος από την άλλη διεργασία.
  - Της *παρατεταμένης στέρησης* (starvation) όπου κάποια διεργασία δεν προλαβαίνει να δεσμεύσει κάποιον πόρο λόγω της πιο γρήγορης δέσμευσής του από άλλες διεργασίες.

5. Συνεργασία μεταξύ διεργασιών

- *Μέσω κοινής χρήσης κάποιων αντικειμένων* (μεταβλητών, αρχείων, βάσεων δεδομένων, κλπ.). Τα προβλήματα του αμοιβαίου αποκλεισμού, του αδιέξοδου και της παρατεταμένης στέρησης υπάρχουν και εδώ. Όμως τα κοινά αντικείμενα μπορούν να προσπελασθούν για διάβασμα ή για γράψιμο και μόνο στη δεύτερη περίπτωση χρειάζεται αμοιβαίος αποκλεισμός. Ένα άλλο πρόβλημα που παρατηρείται εδώ και έχει να κάνει με την ενημέρωση κοινών δεδομένων είναι αυτό της *συνέπειας των δεδομένων* (data coherence) όπως φαίνεται στο ακόλουθο παράδειγμα:

```
P1:   a:=a+1;                    a:=a+1;
      b:=b+1;                    b:=2*b;
P2:   b:=2*b;                    b:=b+1;
      a:=2*a;                    a:=2*a;
```

όπου το ζητούμενο είναι μετά το τέλος εκτέλεσης των εντολών να ισχύει η σχέση `a=b`. Με την εκτέλεση στα αριστερά αυτό επιτυγχάνεται αλλά με την εκτέλεση στα δεξιά δεν επιτυγχάνεται παρόλο που οι δύο διεργασίες έχουν σεβασθεί τη συνθήκη αμοιβαίου αποκλεισμού για τις δύο μεταβλητές ξεχωριστά. Αυτό που χρειάζεται είναι και οι τέσσερεις εντολές να θεωρηθούν κρίσιμο τμήμα.

- *Μέσω άμεσης επικοινωνίας με τη χρήση μηνυμάτων* (messages). Λόγω της έλλειψης κοινού αντικειμένου επικοινωνίας δεν τίθεται θέμα αμοιβαίου αποκλεισμού αλλά τα άλλα προβλήματα παραμένουν. Αδιέξοδο μπορεί να δημιουργηθεί αν μία διεργασία περιμένει μήνυμα από μία άλλη και αντίστροφα. Παρατεταμένη στέρηση μπορεί να προκύψει αν λ.χ. μία διεργασία P1 προσπαθεί να επικοινωνήσει με 2 άλλες διεργασίες P2 και P3 και αυτές με την P1· η P1 μπορεί να επικοινωνεί συνεχώς με την P2 και η P3 να μην καταφέρει ποτέ να ανταλλάξει μηνύματα με την P1.

6. Συμβολισμοί για ταυτόχρονη επεξεργασία

- Για την υποστήριξη ταυτόχρονης επεξεργασίας στο επίπεδο λογισμικού θα πρέπει οι γλώσσες προγραμματισμού να έχουν συμβολισμούς που να δηλώνουν ταυτοχρονία, διαδιεργασιακή επικοινωνία, κρίσιμα τμήματα, κλπ. Κάθε γλώσσα προγραμματισμού έχει τους δικούς της συμβολισμούς αλλά για μία γλώσσα βασισμένη στο δομημένο προγραμματισμό μπορούμε να θεωρήσουμε την ύπαρξη ενός συμβολισμού του τύπου `parbegin…parend` όπου όλες οι εντολές μέσα στο μπλοκ εκτελούνται ταυτόχρονα.

- Επίσης μπορούμε να θεωρήσουμε την υποστήριξη δύο ενσωματωμένων συναρτήσεων `enter_critical(R)` και `exit_critical(R)` για τον προσδιορισμό κρίσιμων τμημάτων αναφορικά με κάποιον πόρο `R`.

- Έτσι το προηγούμενο παράδειγμα θα μπορούσε να γραφεί σε μία (ψευδο) γλώσσα προγραμματισμού που υποστηρίζει τους ανωτέρω συμβολισμούς ως εξής:

```
program mutual_excl;

int a:=1, b:=1;

procedure P1(int a,b)
begin
     enter_critical(a,b);
     a:=a+1;
     b:=b+1;
     exit_critical(a,b);
end;

procedure P2(int a,b)
begin
     enter_critical(a,b);
     b:=2*b;
     a:=2*a;
     exit_critical(a,b);
end;

parbegin
          P1(a,b);
          P2(a,b);
parend

end.
```

7. Αμοιβαίος Αποκλεισμός

- Για τη σωστή υποστήριξη αμοιβαίου αποκλεισμού πρέπει να ικανοποιούνται οι ακόλουθες προϋποθέσεις:
  1. Όλες οι διεργασίες υπόκεινται στον περιορισμό του αμοιβαίου αποκλεισμού, δηλαδή μόνο μία διεργασία ανά πάσα στιγμή μπορεί να βρίσκεται στο κρίσιμο τμήμα της που αντιστοιχεί σε κάποιο συγκεκριμένο πόρο ή κοινό αντικείμενο.
  2. Αν μία διεργασία διακοπεί όταν δεν βρίσκεται σε κρίσιμο τμήμα, θα πρέπει να το κάνει με τρόπο που να μην επηρεάζει άλλες διεργασίες.
  3. Δεν επιτρέπεται η επ' αόριστον αναμονή μίας διεργασίας για να εισέλθει σε κρίσιμο τμήμα, δηλαδή φαινόμενα αδιέξοδου ή παρατεταμένης στέρησης.
  4. Αν μία διεργασία δεν βρίσκεται σε κάποιο κρίσιμο τμήμα δεν μπορεί να απαγορέψει σε άλλη διεργασία την είσοδο στο κρίσιμο τμήμα.
  5. Δεν επιτρέπονται υποθέσεις σε ότι αφορά την ταχύτητα εκτέλεσης των διεργασιών ή το πλήθος των επεξεργαστών στο σύστημα.
  6. Δεν επιτρέπεται η επ' αόριστον παραμονή μίας διεργασίας σε κρίσιμο τμήμα.

- Η υποστήριξη αμοιβαίου αποκλεισμού και η ικανοποίηση των ανωτέρω προϋποθέσεων μπορεί να γίνει σε 4 επίπεδα:
  - Στο επίπεδο του λογισμικού, δηλαδή με τη χρήση κατάλληλων προγραμματιστικών τεχνικών σε κάποια γλώσσα προγραμματισμού.
  - Στο επίπεδο του υλικού, μέσω της υλοποίησης ειδικών εντολών σε γλώσσα μηχανής.
  - Στο επίπεδο του Λ.Σ. με την παροχή από το ίδιο το Λ.Σ. προς τον χρήστη κατάλληλων μηχανισμών.
  - Στο επίπεδο της γλώσσας προγραμματισμού με την παροχή από την ίδια τη γλώσσα προς τον προγραμματιστή κατάλληλων μηχανισμών οι οποίοι υποστηρίζονται από τον μεταγλωττιστή της γλώσσας αυτής.

8. Αμοιβαίος αποκλεισμός σε επίπεδο λογισμικού

- Αλγόριθμος του Dekker. Περίπτωση με 2 διεργασίες:

- ```
  program Dekker1;

  int process_number;

  procedure Process1;
  begin
       while true do
       begin
            while process_number=2 do {nothing} ;
            < κρίσιμο τμήμα >
            process_number:=2;
       end
  end;

  procedure Process2;
  begin
       while true do
       begin
            while process_number=1 do {nothing} ;
            < κρίσιμο τμήμα >
            process_number:=1;
       end
  end;

  begin
       process_number:=1;
       parbegin
                 Process1;
                 Process2;
       parend
  end.
  ```
  ```

8. Αμοιβαίος αποκλεισμός σε επίπεδο λογισμικού (συνέχεια)

- Η συνάρτηση `enter_critical()` υλοποιείται με το βρόχο που ελέγχει συνέχεια την τιμή της μεταβλητής `process_number` περιμένοντας να πάρει σαν τιμή τον αριθμό της διεργασίας. Η συνάρτηση `exit_critical()` υλοποιείται με τη χρήση του τελεστή ανάθεσης που δίνει στη μεταβλητή `process_number` σαν τιμή τον αριθμό της άλλης διεργασίας.

- Απαιτεί την *αυστηρή εναλλαγή* (strict alteration) των διεργασιών κατά την εισαγωγή τους στα κρίσιμα τμήματα με αποτέλεσμα η γρηγορότερη από τις διεργασίες να καθυστερείται από την πιο αργή (από πλευράς χρήσης του κρίσιμου τμήματος) διεργασία. Αν λ.χ. η P1 χρησιμοποιεί την ΚΜΕ μόνο μία φορά την ώρα και η P2 100 φορές, τότε η P2 είναι αναγκασμένη να ακολουθεί τον ρυθμό της πολύ πιο αργής P1, διότι δεν είναι δυνατόν να χρησιμοποιήσει ξανά η P2 την ΚΜΕ πριν το χρησιμοποιήσει η P1.

- Επίσης, η χρήση του βρόχου `while` για τον επαναλαμβανόμενο έλεγχο της μεταβλητής `process_number` δημιουργεί το φαινόμενο της *ενεργούς αναμονής* (busy waiting) που σπαταλά χρόνο της ΚΜΕ.

- Τέλος, το πιο σημαντικό από όλα, αν μία διεργασία βρισκόμενη στο κρίσιμο τμήμα της διακοπεί απρόοπτα, η άλλη δεν μπορεί να συνεχίσει δημιουργώντας αδιέξοδο.

- Η επόμενη παραλλαγή του αλγόριθμου προσπαθεί να απαλείψει αυτά τα προβλήματα.

8. Αμοιβαίος αποκλεισμός σε επίπεδο λογισμικού (συνέχεια)

- ```
  program Dekker2;

  bool P1inside, P2inside;

  procedure Process1;
  begin
       while true do
       begin
            while P2inside do {nothing} ;
            P1inside:=true;
            < κρίσιμο τμήμα >
            P1inside:=false;
       end
  end;

  procedure Process2;
  begin
       while true do
       begin
            while P1inside do {nothing} ;
            P2inside:=true;
            < κρίσιμο τμήμα >
            P2inside:=false;
       end
  end;

  begin
       P1inside:=false; P2inside:=false;
       parbegin
                 Process1;
                 Process2;
       parend
  end.
  ```
  ```

8. Αμοιβαίος αποκλεισμός σε επίπεδο λογισμικού (συνέχεια)

- Η δεύτερη παραλλαγή προσπαθεί να εξαλείψει τα προβλήματα του αδιέξοδου και της αυστηρής εναλλαγής αναγνωρίζοντας ότι χρειάζεται η αποθήκευση πληροφοριών για την κατάσταση και των δύο διεργασιών και όχι μόνο αυτής που βρίσκεται μέσα στο κρίσιμο τμήμα.

- Έτσι χρησιμοποιούνται δύο `boolean` μεταβλητές και κάθε διεργασία μπορεί να ενημερώσει τη δικιά της μεταβλητή αλλά και να δει την τιμή της άλλης μεταβλητής για να πληροφορηθεί την κατάσταση της άλλης διεργασίας.

- Αν και λύνει εν μέρει τα προβλήματα που αναφέρθηκαν, η λύση αυτή δημιουργεί ένα χειρότερο: δεν εγγυάται τον αμοιβαίο αποκλεισμό, δηλαδή υπάρχει περίπτωση και οι δύο διεργασίες να βρεθούν ταυτόχρονα στο κρίσιμο τμήμα.

- Αυτό συμβαίνει αν οι δύο διεργασίες, ταυτόχρονα, έχοντας διαπιστώσει ότι η τιμή της μεταβλητής της άλλης διεργασίας ειναι `false` συνεχίζουν και αφού θέσουν την τιμή `true` στη δικιά τους μπαίνουν στο κρίσιμο τμήμα.

- Το πρόβλημα επομένως έγκειται στο ότι μία διεργασία μπορεί να αλλάξει την κατάσταση τής μεταβλητής της αφού η άλλη την έχει ελέγξει.

- Η τρίτη παραλλαγή προσπαθεί να λύσει το πρόβλημα που αναφέρθηκε προηγουμένως με το να υποχρεώνει κάθε διεργασία να δηλώσει πρώτα ότι επιθυμεί να μπει στο κρίσιμο τμήμα της πριν αρχίσει να εκτελεί το βρόχο (ουσιαστικά εναλλάσουμε τις δύο γραμμές κώδικα).

- Αν και πράγματι αυτή η λύση υποστηρίζει σωστά τον αμοιβαίο αποκλεισμό εν τούτοις επαναφέρει το πρόβλημα του αδιέξοδου: αν προλάβουν και οι δύο διεργασίες να θέσουν την τιμή της αντίστοιχης μεταβλητής τους σε `true` πριν προλάβει η άλλη διεργασία να αρχίσει να εκτελεί το βρόχο `while` τότε η κάθε μία θα νομίζει ότι η άλλη έχει ήδη αρχίσει να εκτελεί το κρίσιμο τμήμα της.

8. Αμοιβαίος αποκλεισμός σε επίπεδο λογισμικού (συνέχεια)

- ```
  program Dekker3;

  bool P1wantstoenter, P2wantstoenter;

  procedure Process1;
  begin
       while true do
       begin
            P1wantstoenter:=true;
            while P2wantstoenter do {nothing} ;
            < κρίσιμο τμήμα >
            P1wantstoenter:=false;
       end
  end;

  procedure Process2;
  begin
       while true do
       begin
            P2wantstoenter:=true;
            while P1wantstoenter do {nothing} ;
            < κρίσιμο τμήμα >
            P2wantstoenter:=false;
       end
  end;

  begin
       P1wantstoenter:=false; P2wantstoenter:=false;
       parbegin
                 Process1;
                 Process2;
       parend
  end.
  ```
  ```

8. Αμοιβαίος αποκλεισμός σε επίπεδο λογισμικού (συνέχεια)

- Το πρόβλημα με την τρίτη παραλλαγή ήταν ότι κάθε διεργασία μπορούσε να επιμείνει να μπει στο κρίσιμο τμήμα της χωρίς να υποχρεούται να κάνει πίσω, δημιουργώντας έτσι αδιέξοδο στην εκτέλεση του βρόχου.

- Η τέταρτη παραλλαγή προσπαθεί να επιλύσει το πρόβλημα με το να αναγκάζει μία διεργασία που έχει δηλώσει ότι θέλει να εκτελέσει το κρίσιμο τμήμα της να κάνει πίσω αν η επιθυμία της δεν έχει ικανοποιηθεί μέσα σε κάποιο μικρό αλλά τυχαίο χρονικό διάστημα.

- Παρ' όλο που φαίνεται ότι η παραλλαγή αυτή λύνει όλα τα προβλήματα χωρίς να δημιουργεί καινούργια, εν τούτοις αυτό δεν αληθεύει. Όσο και αν είναι απίθανο, είναι ωστόσο δυνατόν να δημιουργηθεί το πρόβλημα της *επ' αόριστον αναμονής* (indefinite postponement) αν δημιουργηθεί η ακόλουθη αλληλουχία εκτέλεσης εντολών:

  Η διεργασία `P1` θέτει τη τιμή της μεταβλητής `P1wantstoenter` σαν `true`
  Η διεργασία `P2` θέτει τη τιμή της μεταβλητής `P2wantstoenter` σαν `true`
  Η διεργασία `P1` ελέγχει τη τιμή της μεταβλητής `P2wantstoenter`
  Η διεργασία `P2` ελέγχει τη τιμή της μεταβλητής `P1wantstoenter`
  Η διεργασία `P1` θέτει τη τιμή της μεταβλητής `P1wantstoenter` σαν `false`
  Η διεργασία `P2` θέτει τη τιμή της μεταβλητής `P1wantstoenter` σαν `false`
  Η διεργασία `P1` θέτει τη τιμή της μεταβλητής `P1wantstoenter` σαν `true`
  Η διεργασία `P2` θέτει τη τιμή της μεταβλητής `P2wantstoenter` σαν `true`

- Αυτή η αλληλουχία εκτέλεσης εντολών μπορεί να συνεχίσει επ' άπειρον χωρίς καμμία από τις δύο διεργασίες να μπορέσει ποτέ να εκτελέσει το κρίσιμο τμήμα της.

- Η πέμπτη (!) παραλλαγή χρησιμοποιεί αρκετές από τις τεχνικές που αναφέρθηκαν μέχρι τώρα. Η κατανόηση τού γιατί η τελευταία αυτή παραλλαγή επιλύει όλα τα προβλήματα που αναφέρθηκαν χωρίς να δημιουργεί καινούργια αφήνεται σαν άσκηση.

8. Αμοιβαίος αποκλεισμός σε επίπεδο λογισμικού (συνέχεια)

- ```
  program Dekker4;

  bool P1wantstoenter, P2wantstoenter;

  procedure Process1;
  begin
       while true do
       begin
            P1wantstoenter:=true;
            while P2wantstoenter do
            begin
                 P1wantstoenter:=false;
                 delay(random_number);
                 P1wantstoenter:=true;
            end
            < κρίσιμο τμήμα >
            P1wantstoenter:=false;
       end
  end;

  procedure Process2;
  begin
       while true do
       begin
            P2wantstoenter:=true;
            while P1wantstoenter do
            begin
                 P2wantstoenter:=false;
                 delay(random_number);
                 P2wantstoenter:=true;
            end
            < κρίσιμο τμήμα >
            P2wantstoenter:=false;
       end
  end;

  begin
       P1wantstoenter:=false; P2wantstoenter:=false;
       parbegin
                 Process1;
                 Process2;
       parend
  end.
  ```
  ```

8. Αμοιβαίος αποκλεισμός σε επίπεδο λογισμικού (συνέχεια)

- ```
  program Dekker_final;

  int favoured_process;
  bool P1wantstoenter, P2wantstoenter;

  procedure Process1;
  begin
       while true do
       begin
            P1wantstoenter:=true;
            while P2wantstoenter do
            if favoured_process=2 then
               begin
                     P1wantstoenter:=false;
                     while favoured_process=2 do {nothing} ;
                     P1wantstoenter:=true;
               end
            < κρίσιμο τμήμα >
            favoured_process:=2;
            P1wantstoenter:=false;
       end
  end;

  procedure Process2;
  begin
       while true do
       begin
            P2wantstoenter:=true;
            while P1wantstoenter do
            if favoured_process=1 then
               begin
                     P2wantstoenter:=false;
                     while favoured_process=1 do {nothing} ;
                     P2wantstoenter:=true;
               end
            < κρίσιμο τμήμα >
            favoured_process:=1;
            P2wantstoenter:=false;
       end
  end;

  begin
       P1wantstoenter:=false; P2wantstoenter:=false;
       favoured_process:=1;
       parbegin
                 Process1;
                 Process2;
       parend
  end.
  ```
  ```

8. Αμοιβαίος αποκλεισμός σε επίπεδο λογισμικού (συνέχεια)

- Αλγόριθμος του Peterson (για 2 διεργασίες, πιο απλός από του Dekker):

- ```
  program Peterson;

  int favoured_process;
  bool P1wantstoenter, P2wantstoenter;

  procedure Process1;
  begin
       while true do
       begin
            P1wantstoenter:=true;
            favoured_process:=2;
            while P2wantstoenter and favoured_process=2 do ;
            < κρίσιμο τμήμα >
            P1wantstoenter:=false;
       end
  end;

  procedure Process2;
  begin
       while true do
       begin
            P2wantstoenter:=true;
            favoured_process:=1;
            while P1wantstoenter and favoured_process=1 do ;
            < κρίσιμο τμήμα >
            P2wantstoenter:=false;
       end
  end;

  begin
       P1wantstoenter:=false; P2wantstoenter:=false;
       favoured_process:=1;
       parbegin
                 Process1;
                 Process2;
       parend
  end.
  ```
  ```

9. Αμοιβαίος αποκλεισμός σε επίπεδο υλικού

- Ξεκινώντας από τη σκέψη ότι αυτό που διακόπτει τη λειτουργία μίας διεργασίας είναι οι διακόπτες (interrupts) θα μπορούσαμε να επιτύχουμε αμοιβαίο αποκλεισμό με την αποσύνδεση και επανασύνδεση των διακοπτών. Μία διεργασία θα μπορούσε να έχει κώδικα του τύπου:

```
< αποσύνδεση διακοπτών >
< κρίσιμο τμήμα >
< επανασύνδεση διακοπτών >
```

Αυτή η λύση όμως είναι ασύμφορη λόγω του ότι μειώνει δραματικά την απόδοση του συστήματος. Επιπλέον, δεν είναι εφαρμόσιμη σε συστήματα με πολλούς επεξεργαστές.

- Μία άλλη λύση ξεκινάει από τη σκέψη ότι στο επίπεδο του υλικού η προσπέλαση σε μία συγκεκριμένη θέση μνήμης από κάποια διεργασία εμποδίζει αυτόματα την προσπέλαση στην ίδια θέση μνήμης από άλλη διεργασία. Αυτό επιτρέπει την υποστήριξη μίας εντολής “έλεγξε και θέσε” (test and set) `testset(i)` η οποία αν η τιμή της θέσης μνήμης `i` είναι 0 το κάνει 1 και δηλώνει επιτυχή εκτέλεση και αν η τιμή της `i` είναι 1 απλά δηλώνει αποτυχία. Σημειωτέον ότι ο έλεγχος και αλλαγή της τιμής της `i` γίνονται σαν αδιαίρετες (indivisible) πράξεις:

```
boolean function testset(int i);
begin
     if i=0 then begin i:=1; testset:=true end
            else testset:=false;
end
```

- Μία άλλη παρόμοια λύση είναι αυτή της εντολής `exchange(r,i)` η οποία ανταλλάσει τα περιεχόμενα της θέσης μνήμης `i` με αυτά του καταχωρητή `r`. Και εδώ, όσο διαρκεί η εκτέλεση της εντολής η θέση μνήμης είναι κλειδωμένη. Η υλοποίηση της `exchange(r,i)` είναι απλή:

```
procedure exchange(register r; byte m);
variable temp;
begin
     temp:=m; m:=r; r:=temp;
end
```

9. Αμοιβαίος αποκλεισμός σε επίπεδο υλικού (συνέχεια)

- Κάνοντας χρήση της εντολής `testset` ο αμοιβαίος αποκλεισμός μπορεί να υλοποιηθεί ως ακολούθως:

- ```
  program Mutual_Exclusion;

  int flag;

  procedure Process1;
  begin
       while true do
       begin
            while not testset(flag) do {nothing} ;
            < κρίσιμο τμήμα >
            flag:=0;
       end
  end;

  procedure Process2;
  begin
       while true do
       begin
            while not testset(flag) do {nothing} ;
            < κρίσιμο τμήμα >
            flag:=0;
       end
  end;

  begin
       flag:=0;
       parbegin
                 Process1;
                 Process2;
       parend
  end.
  ```
  ```

9. Αμοιβαίος αποκλεισμός σε επίπεδο υλικού (συνέχεια)

- Κάνοντας χρήση της εντολής `exchange` ο αμοιβαίος αποκλεισμός μπορεί να υλοποιηθεί ως ακολούθως:

```
program Mutual_Exclusion;

int flag;

procedure Process1;
begin
      int key;
      while true do
      begin
            key:=1;
            while key≠0 do exchange(key,flag);
            < κρίσιμο τμήμα >
            exchange(key,flag);
      end
end;

procedure Process2;
begin
      int key;
      while true do
      begin
            key:=1;
            while key≠0 do exchange(key,flag);
            < κρίσιμο τμήμα >
            exchange(key,flag);
      end
end;

begin
      flag:=0;
      parbegin
                  Process1;
                  Process2;
      parend
end.
```

9. Αμοιβαίος αποκλεισμός σε επίπεδο υλικού (συνέχεια)

- Οι ανωτέρω λύσεις έχουν τα εξής πλεονεκτήματα:
  - Επεκτείνονται εύκολα για την περίπτωση περισσοτέρων των δύο διεργασιών ακόμα και στην περίπτωση συστημάτων με πολλούς επεξεργαστές. Μοναδική προϋπόθεση είναι η ύπαρξη κοινής μνήμης.
  - Είναι απλές σαν έννοιες και επομένως εύκολο να ελεγχθεί η σωστή χρήση τους.
  - Μπορούν να υποστηρίξουν πολλαπλά κρίσιμα τμήματα με τη χρήση διαφορετικών μεταβλητών.

- Τα μειονεκτήματα είναι όμως σημαντικά:
  - Παρόλο που υποστηρίζουν αμοιβαίο αποκλεισμό εν τούτοις το κάνουν με τη χρήση ενεργούς αναμονής και άρα τη σπατάλη κύκλων μηχανής.
  - Δεν αποφεύγεται το πρόβλημα της παρατεταμένης στέρησης γιατί κάποια διεργασία μπορεί να περιμένει επ' άπειρον να εκτελέσει το κρίσιμο τμήμα της.
  - Επίσης είναι πιθανή η δημιουργία αποκλεισμού στην περίπτωση που μία διεργασία ενώ βρίσκεται στο κρίσιμο τμήμα της διακοπεί από μία άλλη διεργασία μεγαλύτερης προτεραιότητας. Π.χ., αν η P1 ενώ βρίσκεται σε ένα κρίσιμο τμήμα διακοπεί από την P2 η οποία έχει μεγαλύτερη προτεραιότητα από την P1, τότε αν επιπλέον η P2 προσπαθήσει να εισέλθει στο ίδιο κρίσιμο τμήμα θα έχουμε αποκλεισμό: η P2 δεν θα μπορεί να συνεχίσει αλλά λόγω του ότι έχει μεγαλύτερη προτεραιότητα δεν θα αφήσει και την P1 να συνεχίσει.

10. Αμοιβαίος αποκλεισμός σε επίπεδο Λ.Σ.

- Χρήση σημάτων (signals) — δύο ή περισσότερες διεργασίες μπορούν να συγχρονίσουν την εκτέλεσή τους μέσω της αποστολής και αναμονής λήψης σημάτων. Για την καταμέτρηση των σημάτων χρησιμοποιούνται ειδικές μεταβλητές που λέγονται *σημαφόροι* (semaphores) σε συνδυασμό με δύο θεμελιώδεις λειτουργίες: `signal(s)` και `wait(s)`. Οι σημαφόροι προτάθηκαν από τον E. W. Dijkstra το 1965.

- Η `wait(s)` μειώνει την τιμή του σημαφόρου `s` κατά μία μονάδα. Αν η τιμή του `s` γίνει αρνητική η διεργασία που εκτέλεσε την εντολή αναστέλλει την εκτέλεσή της και μπαίνει σε μία ουρά υπό αναστολή διεργασιών σχετιζόμενη με το σημαφόρο `s`.

- Η `signal(s)` αυξάνει την τιμή του σημαφόρου `s` κατά μία μονάδα. Αν η τιμή του `s` δεν είναι θετική τότε μία από τις διεργασίες που βρίσκονται υπό αναστολή στην ουρά τής `s` αλλάζει την κατάστασή της σε έτοιμη για εκτέλεση. Αν ο ορισμός του σημαφόρου καθορίζει ότι η διεργασία που θα ενεργοποιηθεί είναι η πρώτη στην ουρά (δηλαδή αυτή που βρίσκεται υπό αναστολή το μεγαλύτερο χρονικό διάστημα), τότε ο σημαφόρος αυτός λέγεται *ισχυρός*. Στην αντίθετη περίπτωση που δεν καθορίζεται η σειρά ενεργοποίσης των διεργασιών, ο σημαφόρος αυτός λέγεται *αδύνατος*.

- Οι *δυαδικοί σημαφόροι* (binary semaphores) μπορούν να πάρουν μόνο τις τιμές 0 και 1, είναι πιο εύκολο να υλοποιηθούν από τους γενικούς σημαφόρους και μπορεί να αποδειχθεί ότι έχουν την ίδια εκφραστική ικανότητα με τους γενικούς σημαφόρους.

- Εξυπακούεται ότι η εκτέλεση των εντολών `signal` και `wait` θα πρέπει να γίνεται σαν *ατομική ενέργεια* (atomic action). Για την υλοποίησή τους μπορεί να χρησιμοποιηθούν οι τεχνικές που αναφέρθηκαν μέχρι τώρα (αλγόριθοι τύπου Dekker ή Peterson, εντολές τύπου `testset`, κλπ.).

- Οι σημαφόροι προσφέρουν ένα σχετικά εύκολο τρόπο συγχρονισμού και διαδιεργασιακής επικοινωνίας αλλά θέλει προσοχή η χρήση τους γιατί μικρά λάθη μπορούν να οδηγήσουν εύκολα σε δημιουργία αδιεξόδων.

10. Αμοιβαίος αποκλεισμός σε επίπεδο Λ.Σ. (συνέχεια)

- Ορισμός σημαφόρων:

```
type semaphore = record
                    count: integer;
                    queue: list of process;
                 end;
var s: semaphore;

wait(s):
   s.count:=s.count-1;
   if s.count<0
      then begin
              βάλε αυτή τη διεργασία στο s.queue;
              θέσε τη διεργασία υπό αναστολή;
           end;

signal(s):
   s.count:=s.count+1;
   if s.count≤0
      then begin
              αφαίρεσε μία διεργασία P από την s.queue;
              τοποθέτησε την P στη λίστα διεργασιών
              έτοιμων για εκτέλεση;
           end;
```

10. Αμοιβαίος αποκλεισμός σε επίπεδο Λ.Σ. (συνέχεια)

- Ορισμός δυαδικών σημαφόρων:

```
type binary semaphore = record
                        value: (0, 1);
                        queue: list of process;
                    end;
var s: binary semaphore;

waitB(s):
   if s.value=1
      then s.value=0
      else begin
               βάλε αυτή τη διεργασία στο s.queue;
               θέσε τη διεργασία υπό αναστολή;
           end;

signalB(s):
   if s.queue is empty
      then s.value=1
      else begin
               αφαίρεσε μία διεργασία P από την s.queue;
               τοποθέτησε την P στη λίστα διεργασιών
               έτοιμων για εκτέλεση;
           end;
```

10. Αμοιβαίος αποκλεισμός σε επίπεδο Λ.Σ. (συνέχεια)

- Υλοποίηση σημαφόρων με βάση την εντολή `testset`:

```
type semaphore = record
                    count: integer;
                    flag: integer;
                    queue: list of process;
                 end;
var s: semaphore;

wait(s):
   repeat {nothing} until testset(s.flag);
   s.count:=s.count-1;
   if s.count<0
      then begin
              βάλε αυτή τη διεργασία στο s.queue;
              θέσε τη διεργασία υπό αναστολή
              (περιλαμβάνει την εντολή s.flag:=0);
           end;
      else s.flag:=0;

signal(s):
   repeat {nothing} until testset(s.flag);
   s.count:=s.count+1;
   if s.count≤0
      then begin
              αφαίρεσε μία διεργασία P από την s.queue;
              τοποθέτησε την P στη λίστα διεργασιών
              έτοιμων για εκτέλεση;
           end;
   s.flag:=0;
```

10. Αμοιβαίος αποκλεισμός σε επίπεδο Λ.Σ. (συνέχεια)

- Εναλλακτικός τρόπος υλοποίησης σημαφόρων με διακόπτες.

```
wait(s):
   απενεργοποίησε τους διακόπτες;
   s.count:=s.count−1;
   if s.count<0
      then begin
               βάλε αυτή τη διεργασία στο s.queue;
               θέσε τη διεργασία υπό αναστολή και
               ενεργοποίησε τους διακόπτες;
           end
      else ενεργοποίησε τους διακόπτες;

signal(s):
   απενεργοποίησε τους διακόπτες;
   s.count:=s.count+1;
   if s.count≤0
      then begin
               αφαίρεσε μία διεργασία P από την s.queue;
               τοποθέτησε την P στη λίστα διεργασιών
               έτοιμων για εκτέλεση;
           end;
   ενεργοποίησε τους διακόπτες;
```

10. Αμοιβαίος αποκλεισμός σε επίπεδο Λ.Σ. (συνέχεια)

- Κάνοντας χρήση σημαφόρων ο αμοιβαίος αποκλεισμός μπορεί να υλοποιηθεί ως ακολούθως:

```
program Mutual_Exclusion;

semaphore sema;

procedure Process1;
begin
     while true do
     begin
          wait(sema);
          < κρίσιμο τμήμα >
          signal(sema);
     end
end

procedure Process2;
begin
     while true do
     begin
          wait(sema);
          < κρίσιμο τμήμα >
          signal(sema);
     end
end

begin
     sema:=1;
     parbegin
               Process1;
               Process2;
     parend
end
```

10. Αμοιβαίος αποκλεισμός σε επίπεδο Λ.Σ. (συνέχεια)

- Κάνοντας χρήση δύο σημαφόρων μπορούμε να υλοποιήσουμε το κλασσικό πρόβλημα του παραγωγού-καταναλωτή (producer-consumer), για την περίπτωση προσωρινής μνήμης (buffer) μίας θέσης. Μπορούν να δημιουργηθούν παραλλαγές του προβλήματος με προσωρινή μνήμη περισσότερων της μίας θέσεων.

```
program Producer_Consumer;

int buffer;
semaphore value_put, value_get;

procedure Producer;
begin
     int result;
     while true do
     begin
          < υπολόγισε το result >
          wait(value_get);
          buffer:=result;
          signal(value_put);
     end
end

procedure Consumer;
begin
     int result;
     while true do
     begin
          wait(value_put);
          result:=buffer;
          signal(value_get);
          < κατανάλωσε το result >
     end
end

begin
     value_put:=0; value_get:=1;
     parbegin
               Producer;
               Consumer;
     parend
end
```

10. Αμοιβαίος αποκλεισμός σε επίπεδο Λ.Σ. (συνέχεια)

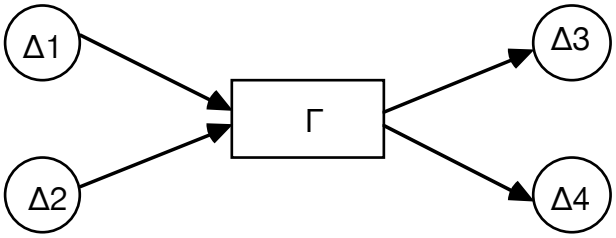
- Ένας άλλος τρόπος είναι τα μηνύματα (messages) που επιτυγχάνουν ταυτόχρονα και το *συντονισμό* και την *επικοινωνία* μεταξύ των διεργασιών.

- Υπάρχουν και εδώ δύο θεμελειώδεις λειτουργίες:
  — `send(destination,message)`
  στέλνει το μήνυμα σε καθορισμένο προορισμό,
  — `receive(source,message)`
  δέχεται μήνυμα από συγκεκριμένη πηγή.

- Ανάλογα με την περίπτωση, η εκτέλεση από μία διεργασία της εντολής `send(destination,message)` μπορεί να προκαλέσει ή όχι την αναστολή εκτέλεσης της διεργασίας μέχρις ότου το μήνυμα φτάσει στον προορισμό του (blocking ή nonblocking `send`).

- Στην περίπτωση εκτέλεσης της `receive`, συνήθως η διεργασία αναστέλλει την εκτέλεσή της μέχρις ότου ληφθεί το ζητούμενο μήνυμα. Στην περίπτωση που και οι δύο εντολές οδηγούν σε αναστολή της διεργασίας που τις κάλεσε μέχρι την αποπεράτωση εκτέλεσής τους, έχουμε την τεχνική του ραντεβού (όπως στη γλώσσα Ada).

- Η επικοινωνία μπορεί να γίνει είτε άμεσα χρησιμοποιώντας σαν `source` και `destination` τις ίδιες τις διεργασίες είτε έμμεσα με τη χρήση κάποιου κοινού “γραμματοκιβωτίου” (mailbox) και εντολών του τύπου
  `send(mailbox,message)`
  `receive(mailbox,message)`



10. Αμοιβαίος αποκλεισμός σε επίπεδο Λ.Σ. (συνέχεια)

- Κάνοντας χρήση αποστολής και λήψης μηνυμάτων μέσω ενός γραμματοκιβωτίου μπορούμε να μοντελοποιήσουμε τον αμοιβαίο αποκλεισμό ως εξής:

```
program mutual_exclusion;

mailbox box;

procedure Process1;
begin
     message msg;
     while true do
     begin
          receive(box,msg);
          < κρίσιμο τμήμα >
          send(box,msg);
     end
end

procedure Process2;
begin
     message msg;
     while true do
     begin
          receive(box,msg);
          < κρίσιμο τμήμα >
          send(box,msg);
     end
end

begin
     send(box,null);
     parbegin
               Process1;
               Process2;
     parend
end
```

10. Αμοιβαίος αποκλεισμός σε επίπεδο Λ.Σ. (συνέχεια)

- Κάνοντας χρήση δύο γραμματοκιβωτίων μπορούμε να μοντελοποιήσουμε το πρόβλημα του παραγωγού-καταναλωτή ως εξής:

```
program Producer_Consumer;

mailbox prod, cons;

procedure Producer;
begin
      message msg;
      while true do
      begin
            receive(prod,msg);
            msg:=produced_value;
            send(cons,msg);
      end
end

procedure Consumer;
begin
      message msg;
      while true do
      begin
            receive(cons,msg);
            < κατανάλωσε το msg >
            send(prod,null);
      end
end

begin
      send(prod,null);
      parbegin
                  Producer;
                  Consumer;
      parend
end
```

11. Αμοιβαίος αποκλεισμός σε επίπεδο γλωσσών προγραμματισμού

- Έχει το πλεονέκτημα ότι ο προγραμματισμός γίνεται σε υψηλότερο επίπεδο και ελαχιστοποιεί τις πιθανότητες για λάθη. Μία από τις πιο γνωστές αρχές συγχρονισμού σε επίπεδο γλωσσών προγραμματισμού είναι ο *παρακολουθητής* (monitor) που προτάθηκε από τους Tony Hoare και Brinch Hansen το 1974-75. Παρακολουθητής είναι μία ομάδα από διαδικασίες, μεταβλητές και δομές δεδομένων που ομαδοποιούνται σε ένα ειδικό τύπο πακέτου. Μία διεργασία μπορεί να καλέσει τις διαδικασίες ενός παρακολουθητή αλλά όχι και να προσπελάσει άμεσα στις εσωτερικές δομές του πακέτου οι οποίες θεωρούνται προστατευόμενοι πόροι (καθιστώντας έτσι τον παρακολουθητή προπομπό του αντικειμενοστρεφούς προγραμματισμού).

- Ένα από τα πλεονεκτήματα της υψηλού επιπέδου αυτής προσέγγισης είναι ότι υπεύθυνος για την υλοποίηση του αμοιβαίου αποκλεισμού είναι ο μεταγλωττιστής της γλώσσας προγραμματισμού που υποστηρίζει παρακολουθητές και όχι ο προγραμματιστής. Σε χαμηλότερο επίπεδο βεβαίως η υλοποίηση γίνεται με πιο στοιχειώδεις μηχανισμούς όπως δυαδικούς σημαφόρους, κλπ.

- Ο συγχρονισμός επιτυγχάνεται μέσω των εξής μηχανισμών:
  - — Μόνο μία διεργασία κάθε χρονική στιγμή μπορεί να είναι ενεργή σε έναν παρακολουθητή.
  - — Αναστολή και επανεργοποίηση μίας διεργασίας γίνεται με τις εντολές `wait(s)` και `signal(s)` όπου `s` είναι μία *μεταβλητή συνθήκης* (condition variable).

- Η `wait(s)` αναστέλλει την καλούσα διεργασία, επιτρέποντας έτσι σε άλλη διεργασία να χρησιμοποιήσει τον παρακολουθητή. Η `signal(s)` επιλέγει, με βάση την πολιτική που ακολουθεί ο χρονοδρομολογητής, μία από τις διεργασίες που έχουν αναστελλεί (αν υπάρχουν — αλλιώς η `signal` δεν κάνει τίποτα) και περιμένουν στην ουρά της `s`, την επανεργοποιεί και επίσης υποχρεώνει τη διεργασία που εκτέλεσε τη `signal` να εξέλθει του παρακολουθητή.

11. Αμοιβαίος αποκλεισμός σε επίπεδο γλωσσών προγραμματισμού (συνέχεια)

- Κάνοντας χρήση παρακολουθητών μπορούμε να υλοποιήσουμε το πρόβλημα του παραγωγού-καταναλωτή για μία προσωρινή θέση μνήμης:

```
monitor Prod_Cons;
      condition full, empty;
      integer count=0;

      procedure enter;
      begin
            if count=1 then wait(full);
            enter_item;
            count:=1;
            signal(empty);
      end

      procedure remove;
      begin
            if count=0 then wait(empty);
            remove_item;
            count:=0;
            signal(full);
      end
endmonitor

procedure Producer;
begin
      while true do
      begin
            produce_item;
            Prod_Cons.enter;
      end
end

procedure Consumer;
begin
      while true do
      begin
            Prod_Cons.remove;
            consume_item;
      end
end
```

11. Αμοιβαίος αποκλεισμός σε επίπεδο γλωσσών προγραμματισμού (συνέχεια)

- Αν και η χρήση των εντολών `wait` και `signal` σε έναν παρακολουθητή είναι παρόμοια της χρήσης των αντίστοιχων εντολών ενός σημαφόρου, εν τούτοις ο παρακολουθητής έχει μερικά σημαντικά προτερήματα (και διαφορές) έναντι του σημαφόρου:
  - Οι εντολές `wait` και `signal` ενός σημαφόρου είναι διασκορπισμένες μέσα στον κώδικα ενός προγράμματος, καθιστώντας έτσι την αποσφαλμάτωση αυτού του τελευταίου δύσκολη. Αντίθετα, η χρήση των αντίστοιχων εντολών σε έναν παρακολουθητή είναι περιορισμένη μέσα στον κώδικα που τον ορίζει.
  - Λόγω του γεγονότος ότι οι δομές δεδομένων μέσα σε έναν παρακολουθητή δεν είναι άμεσα προσπελάσιμες από τις διεργασίες που κάνουν χρήση τους, υποστηρίζεται αυτόματα ο αμοιβαίος αποκλεισμός και ο προγραμματιστής χρειάζεται να ασχοληθεί μόνο με τον σωστό συγχρονισμό εναλλαγής των διεργασιών στην πρόσβασή τους στις δομές αυτές. Αντίθετα, όταν χρησιμοποιούνται σημαφόροι, ο προγραμματιστής έχει την επιπλέον υποχρέωση να υλοποιήσει σωστά και τον αμοιβαίο αποκλεισμό στην πρόσβαση σε κοινούς πόρους.
  - Λόγω του ότι οι μηχανισμοί συγχρονισμού των διεργασιών που χρησιμοποιούν έναν παρακολουθητή περιορίζονται μέσα σε αυτόν, από τη στιγμή που ο παρακολουθητής έχει υλοποιηθεί σωστά πρόσβαση στα κρίσιμα τμήματά του γίνεται με τον ενδεδειγμένο τρόπο από οποιαδήποτε διεργασία. Αντίθετα, στην περίπτωση χρήσης σημαφόρων θα πρέπει όλες οι διεργασίες που έχουν πρόσβαση σε κάποιο κρίσιμο τμήμα να προγραμματισθούν σωστά.
  - Σημειωτέον, ότι η εντολή `signal` σε έναν παρακολουθητή λειτουργεί διαφορετικά από την αντίστοιχη εντολή ενός (γενικού) σημαφόρου. Στη δεύτερη περίπτωση ακόμα και αν δεν υπάρχει καμμία διεργασία υπό αναστολή αναφορικά με κάποιον σημαφόρο, εκτελώντας ένα `signal` για αυτόν οδηγεί στην αύξηση της τιμής του κατά μία μονάδα. Στην πρώτη περίπτωση η εκτέλεση της `signal` για κάποια μεταβλητή συνθήκης απλά αγνοείται αν δεν υπάρχει καμμία διεργασία υπό αναστολή αναφορικά με τη συνθήκη αυτή.

11. Αμοιβαίος αποκλεισμός σε επίπεδο γλωσσών προγραμματισμού (συνέχεια)

- Ο ορισμός ενός παρακολουθητή όπως διαμορφώθηκε από τους Hoare και Hansen επιβάλλει ότι όταν εκτελείται μία εντολή `signal(s)` θα πρέπει *αμέσως* να ενεργοποιηθεί και να αρχίσει εκτέλεση κάποια από τις διεργασίες που είναι υπό αναστολή στη συνθήκη `s`. Επομένως η διεργασία που εκτέλεσε την `signal` θα πρέπει ή να τερματίσει αμέσως την εκτέλεσή της ή να την αναστείλει. Αυτό δημιουργεί ορισμένα προβλήματα:
  - Αν η διεργασία που εκτέλεσε την `signal` δεν έχει ολοκληρώσει την εκτέλεσή της τότε θα χρειασθούν δύο επιπλέον εναλλαγές περιβάλλοντος για αυτήν (αναστολή και επανεργοποίηση).
  - Ο μηχανισμός εναλλαγής στην ΚΜΕ της διεργασίας που εκτέλεσε την `signal(s)` με κάποια άλλη διεργασία υπό αναστολή στην `s` πρέπει να γίνει με ακρίβεια διαφορετικά κάποια τρίτη διεργασία μπορεί να προλάβει να εκτελεσθεί πρώτη και να αλλάξει την `s`.

- Οι Lampson και Redell πρότειναν ένα διαφορετικό ορισμό του παρακολουθητή όπου αντί για την `signal` υπάρχει η εντολή `notify` με την εξής ερμηνεία: η εκτέλεση της εντολής `notify(s)` ναι μεν μεταφέρει το μήνυμα στην ουρά των υπό αναστολή διεργασιών στη συνθήκη `s` ότι κάποια από αυτές θα πρέπει κάποια στιγμή να ενεργοποιηθεί αλλά συνεχίζει την εκτέλεση της διεργασίας που έκανε χρήση της `notify`. Η διεργασία που πρέπει να ενεργοποιηθεί θα επαναρχίσει εκτέλεση σε κάποια κατάλληλη για το Λ.Σ. μελλοντική στιγμή. Έτσι επιτυγχάνονται τα εξής:
  - Τα προηγούμενα μειονεκτήματα εξαλείφονται αφού δεν χρειάζονται οι δύο επιπλέον εναλλαγές περιβάλλοντος για τη διεργασία που εκτέλεσε την `notify` και επιπλέον το Λ.Σ. δεν είναι υποχρεωμένο να υποστηρίξει αυστηρή εναλλαγή των εμπλεκόμενων διεργασιών.
  - Η μέθοδος αυτή τείνει να δημιουργεί λιγότερα λάθη γιατί δεν μπορεί ο προγραμματιστής να βασισθεί σε υποστήριξη αυστηρής εναλλαγής και είναι υποχρεωμένος να εξετάζει πιο συχνά τις συνθήκες αναστολής των διεργασιών. Επιπλέον οδηγεί στη δημιουργία πιο αρθρωτών (modular) προγραμμάτων.

11. Αμοιβαίος αποκλεισμός σε επίπεδο γλωσσών προγραμματισμού (συνέχεια)

- Με τον παρακολουθητή των Lampson και Redell στο προηγούμενο πρόγραμμα τα `if` πρέπει να αλλαχθούν σε `while`:

- ```
  monitor Prod_Cons;
        condition full, empty;
        integer count=0;

        procedure enter;
        begin
              while count=1 then wait(full);
              enter_item;
              count:=1;
              notify(empty);
        end

        procedure remove;
        begin
              while count=0 then wait(empty);
              remove_item;
              count:=0;
              notify(full);
        end
  endmonitor

  procedure Producer;
  begin
        while true do
        begin
              produce_item;
              Prod_Cons.enter;
        end
  end

  procedure Consumer;
  begin
        while true do
        begin
              Prod_Cons.remove;
              consume_item;
        end
  end
  ```
  ```