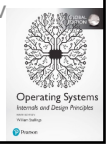
ΕΠΛ222: Λειτουργικά Συστήματα

(μετάφραση στα ελληνικά των διαφανειών του βιβλίου Operating Systems: Internals and Design Principles, 9/E, William Stallings)

Ενότητα 4 (Κεφάλαιο 5 & Παράρτημα Α) Συντρέχων Προγραμματισμός

Οι διαφάνειες αυτές έχουν συμπληρωματικό και επεξηγηματικό χαρακτήρα και σε καμία περίπτωση δεν υποκαθιστούν το βιβλίο

Γιώργος Α. Παπαδόπουλος
Τμήμα Πληροφορικής
Πανεπιστήμιο Κύπρου



1

Περιεχόμενα

- Ταυτοχρονισμός, συντρέχων προγραμματισμός και η έννοια του αμοιβαίου αποκλεισμού.
- Υλοποίηση αμοιβαίου αποκλεισμού:
  - Σε επίπεδο λογισμικού.
  - Σε επίπεδο υλικού.
  - Σε επίπεδο λειτουργικού συστήματος.
  - Σε επίπεδο γλωσσών προγραμματισμού.
- Κλασσικά προβλήματα ταυτοχρονισμού.

2

2

Ταυτοχρονισμός και Παραλληλισμός

- 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 από μία διεργασίες.

3

3

Σενάριο ταυτοχρονισμού

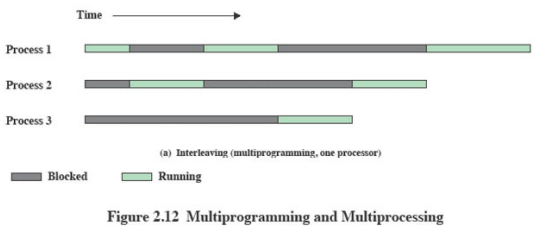


Figure 2.12 Multiprogramming and Multiprocessing

4

4

Σενάριο παραλληλισμού

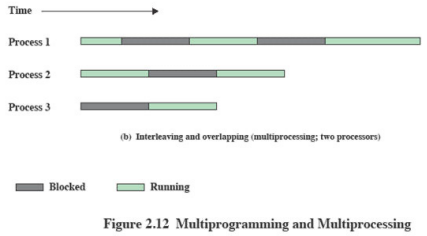


Figure 2.12 Multiprogramming and Multiprocessing

5

5

Λόγοι ανάγκης για ταυτόχρονη εκτέλεση διεργασιών

- Πολυπρογραμματισμός (multiprogramming): Η διαχείριση πολλαπλών διεργασιών σε ένα σύστημα με έναν επεξεργαστή.
- Πολυεπεξεργασία (multiprocessing): Η διαχείριση πολλαπλών διεργασιών σε ένα σύστημα με πολλούς επεξεργαστές.
- Κατανεμημένη επεξεργασία (distributed processing): Η διαχείριση πολλαπλών διεργασιών σε ένα σύστημα με πολλαπλούς Η/Υ, συνδεδεμένους μεταξύ τους με κάποιο δίκτυο.

6

6

Θέματα σχετιζόμενα με τον ταυτοχρονισμό

- Επικοινωνία μεταξύ των διεργασιών.
- Διαμοιρασμός και ανταγωνιστικότητα στη χρήση των πόρων του συστήματος (όπως μνήμη, αρχεία, συσκευές).
- Συγχρονισμός μεταξύ των διεργασιών.
- Κατανομή του χρόνου της (ή των) ΚΜΕ στις διεργασίες.

7

Πηγές ταυτοχρονίας

- Ταυτόχρονη εκτέλεση πολλαπλών εφαρμογών με διαμοιρασμό του χρόνου της ΚΜΕ μεταξύ τους.
- Ο δομημένος προγραμματισμός επιτρέπει το σχεδιασμό ενός προγράμματος ως μία ομάδα από ταυτόχρονα εκτελούμενες διεργασίες.
- Το ίδιο το Λ.Σ. αποτελείται από μία μεγάλη ομάδα από ταυτόχρονα εκτελούμενες διεργασίες ή/και νήματα.

8

Συντρέχων προγραμματισμός

- 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.

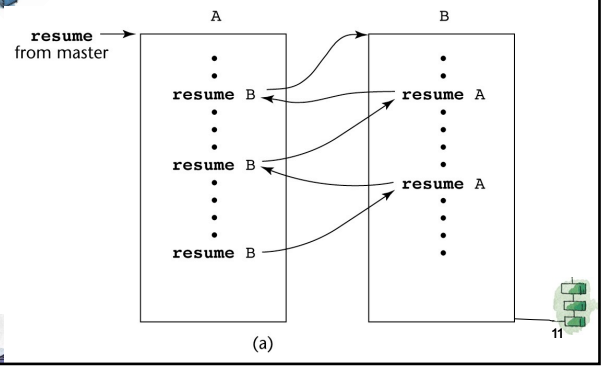
9

Συρρουτίνες

- 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 επανάρχιζε 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

10

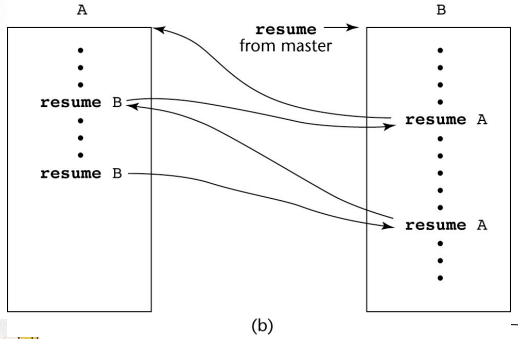
Ο μηχανισμός των συρρουτινών με χρήση της εντολής RESUME — 1



(a)

11

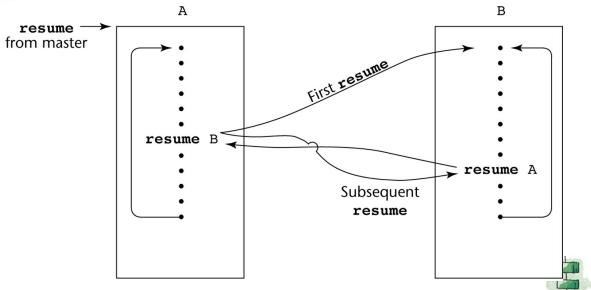
Ο μηχανισμός των συρρουτινών με χρήση της εντολής RESUME — 2



(b)

12

Ο μηχανισμός των συρροutινών με χρήση της εντολής RESUME — 3



13

13

Πρόβλημα συντονισμού αριθμού διαδικασιών

- 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 αντικατασταθούν με ένα # (ο τελευταίος χαρακτήρας δεν μπορεί να είναι *).
- Σειριακή λύση με 3 διαδικασίες:
  - Διαδικασία 1η: Διάβασε *όλες* τις κάρτες σε προσωρινή μνήμη (buffer).
  - Διαδικασία 2η: Κάνε τις αντικαταστάσεις.
  - Διαδικασία 3η: Τύπωσε το αποτέλεσμα σε γραμμές των 125 χαρακτήρων.
- Μειονεκτήματα: Χάσιμο χρόνου (overhead) λόγω του ότι εκτελούνται πολλές εντολές E/E αλλά και μεγάλη σπατάλη μνήμης.

14

14

Επίλυση με συρροutίνες

```
char rs, sp;
char inbuf[80], outbuf[125];

void read()
{
while (1) {
   READCARD (inbuf);
   for (int i=0; i < 80; i++) {
      rs = inbuf [i];
      RESUME squash;
   }
   rs = " ";
   RESUME squash;
 }
}

void print()
{
 while (1) {
    for (int j = 0; j < 125; j++) {
        outbuf [j] = sp;
        RESUME squash;
    }
    OUTPUT (outbuf);
 }
}

void squash()
{
 while (1) {
    if (rs != "*") {
       sp = rs;
       RESUME print;
    }
    else {
       RESUME read;
       if (rs == "*") {
          sp = "#";
          RESUME print;
       }
       else {
          sp = "*";
          RESUME print;
          sp = rs;
          RESUME print;
       }
    }
 }
}

RESUME read;
```

15

15

Προβλήματα που δημιουργεί ο ταυτοχρονισμός (και ανάλογες ευθύνες για το Λ.Σ.)

- Διαχείριση των εκτελούμενων διεργασιών.
- Βέλτιστη κατανομή των διαθέσιμων πόρων αναφορικά με ΚΜΕ, κύρια και περιφερειακή μνήμη και συσκευές E/E 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.
- Δυσκολία αποσφαλμάτωσης (debugging) των προγραμμάτων γιατί συνήθως η εκτέλεση των διεργασιών και η εν γένει συμπεριφορά του συστήματος είναι μη προκαθορισμένη.

16

16

Αλληλεπίδραση μεταξύ διεργασιών

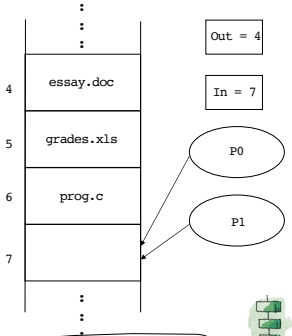
- Η ταυτόχρονη εκτέλεση περισσοτέρων της μίας διεργασιών οδηγεί συχνά σε φαινόμενα *αλληλεπίδρασης* μεταξύ των διεργασιών αυτών (process interaction) που μπορούν να χωρισθούν σε 3 κατηγορίες ανάλογα με το βαθμό στον οποίο μία διεργασία γνωρίζει την ύπαρξη άλλων διεργασιών και πως τυχόν εκμεταλλεύεται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

17

17

Ανταγωνισμός μεταξύ διεργασιών

Η διεργασία `P0` 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 `P1`, 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 `P0` 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 `P1` 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).



18

18

Αμοιβαίος αποκλεισμός

- 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/περιοχές (critical sections/regions) και για τη σωστή εκτέλεση συντρέχουσων διεργασιών πρέπει να διασφαλιστεί ότι *ποτέ δύο ή περισσότερες διεργασίες δεν θα βρίσκονται ταυτόχρονα στο ίδιο κρίσιμο τμήμα*.

Προβλήματα σχετιζόμενα με τον αμοιβαίο αποκλεισμό

- Αδιέξοδο (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.
- Παρατεταμένη στέρησης (starvation), όπου κάποια διεργασία δεν προλαβαίνει να δεσμεύσει κάποιον πόρο λόγω της πιο γρήγορης δέσμευσής του από άλλες διεργασίες.

Έμμεση 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:

```
P0: a = a+1;                    a = a+1; (P0)
    b = b+1;                    b = 2*b; (P1)
P1: b = 2*b;                    b = b+1; (P0)
    a = 2*a;                    a = 2*a; (P1)
```

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.

Άμεση συνεργασία μεταξύ διεργασιών

- 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 `P0` προσπαθεί να επικοινωνήσει με δύο άλλες διεργασίες `P1` και `P2` και αυτές με την `P0`· η `P0` μπορεί να επικοινωνεί συνεχώς με την `P1` και η `P2` να μην καταφέρει ποτέ να ανταλλάξει μηνύματα με την `P0`.

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`.

Γενική μοντελοποίηση αμοιβαίου αποκλεισμού

```
/* PROCESS 0 */

void P0()
{
 while (1)
 {
/* preceding code */ ;
   enter_critical(R);
/* critical section */ ;
   exit_critical(R);
/* following code */ ;
 }
}

/* PROCESS 1 */

void P1()
{
 while (1)
 {
/* preceding code */ ;
   enter_critical(R);
/* critical section */ ;
   exit_critical(R);
/* following code */ ;
 }
}

/* PROCESS n */

void Pn()
{
 while (1)
 {
/* preceding code */ ;
   enter_critical(R);
/* critical section */ ;
   exit_critical(R);
/* following code */ ;
 }
}

parbegin
   P0(); P1(); …; Pn();
parend
```

Εφαρμογή του μοντέλου στο σενάριο της άμεσης συνεργασίας μεταξύ διεργασιών

```
main()
{
   int a = 1, b = 1;

   void P0(int a,b)                 void P1(int a,b)
   {                                {
    enter_critical(a,b);             enter_critical(a,b);
    a = a+1;                         b = 2*b;
    b = b+1;                         a = 2*a;
    exit_critical(a,b);              exit_critical(a,b);
   }                                }

   parbegin
      P0(a,b);
      P1(a,b);
   parend
}
```


25

Σωστή υλοποίηση του αμοιβαίου αποκλεισμού

- Για τη σωστή υποστήριξη του αμοιβαίου αποκλεισμού πρέπει να ικανοποιούνται οι ακόλουθες προϋποθέσεις:
  - 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.
  - Αν μία διεργασία διακοπεί όταν δεν βρίσκεται σε κρίσιμο τμήμα, θα πρέπει να το κάνει με τρόπο που να μην επηρεάζει άλλες διεργασίες.
  - Δεν επιτρέπεται η επ' αόριστον αναμονή μίας διεργασίας για να εισέλθει σε κρίσιμο τμήμα, δηλαδή φαινόμενα αδιέξοδου ή παρατεταμένης στέρησης.
  - Αν μία διεργασία δεν βρίσκεται σε κάποιο κρίσιμο τμήμα δεν μπορεί να απαγορέψει σε άλλη διεργασία την είσοδο στο κρίσιμο τμήμα.
  - Δεν επιτρέπονται υποθέσεις σε ότι αφορά την ταχύτητα εκτέλεσης των διεργασιών ή το πλήθος των επεξεργαστών στο σύστημα.
  - Δεν επιτρέπεται η επ' αόριστον παραμονή μίας διεργασίας σε κρίσιμο τμήμα.

26

Περιεχόμενα

- Ταυτοχρονισμός, συντρέχων προγραμματισμός και η έννοια του αμοιβαίου αποκλεισμού.
- Υλοποίηση αμοιβαίου αποκλεισμού:
  - Σε επίπεδο λογισμικού.
  - Σε επίπεδο υλικού.
  - Σε επίπεδο λειτουργικού συστήματος.
  - Σε επίπεδο γλωσσών προγραμματισμού.
- Κλασσικά προβλήματα ταυτοχρονισμού.

27

Υλοποίηση αμοιβαίου αποκλεισμού σε επίπεδο λογισμικού

- Η υλοποίηση του αμοιβαίου αποκλεισμού γίνεται με τη χρήση κατάλληλων τεχνικών προγραμματισμού σε οποιαδήποτε γλώσσα προγραμματισμού.
- Δεν προϋποθέτει την ύπαρξη στο σύστημα ειδικών μηχανισμών υποστήριξης του αμοιβαίου αποκλεισμού.
- Οι τεχνικές αυτές είναι γνωστές ως οι αλγόριθμοι των Dekker και Peterson. Ο αλγόριθμος του Ολλανδού μαθηματικού Dekker χρησιμοποιήθηκε από τον E. W. Dijkstra το 1965.
- Βασίζονται στο γεγονός ότι το σύστημα κατά κανόνα υποστηρίζει μία στοιχειώδη μορφή αμοιβαίου αποκλεισμού στο επίπεδο πρόσβασης σε κάποια θέση μνήμης, δηλαδή ότι ανά πάσα στιγμή μόνο μία αίτηση για διάβασμα ή γράψιμο σε κάποια θέση μνήμης μπορεί να εξυπηρετηθεί και αν τυχόν υπάρχουν και άλλες τέτοιες αιτήσεις για αυτή τη θέση μνήμης πρέπει να σειριοποιηθούν.
- Η παρουσίαση των αλγόριθμων γίνεται για την περίπτωση 2 διεργασιών αλλά γενικεύονται και για την περίπτωση οποιουδήποτε πλήθους N διεργασιών.

28

Αλγόριθμος του Dekker: Πρώτη προσπάθεια

```
int turn = 0;

void P0()                           void P1()
{                                   {
 while (1) {                         while (1) {
    /* preceding code */ ;              /* preceding code */ ;
    while (turn != 0)                   while (turn != 1)
       /* do nothing */ ;                  /* do nothing */ ;
    /* critical section */ ;            /* critical section */ ;
    turn = 1;                           turn = 0;
    /* following code */ ;              /* following code */ ;
 }                                   }
}                                   }

void main()
{
 parbegin P0(); P1(); parend
}
```


29

Προβλήματα με την πρώτη προσέγγιση

- 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 `P0` χρησιμοποιεί την ΚΜΕ μόνο μία φορά την ώρα και η `P1` 1000 φορές, τότε η `P1` είναι αναγκασμένη να ακολουθεί τον ρυθμό της πολύ πιο αργής `P0`, διότι δεν είναι δυνατόν να χρησιμοποιήσει ξανά η `P1` την ΚΜΕ πριν το χρησιμοποιήσει η `P0`.
- Αν μία από τις διεργασίες διακοπεί λόγω κάποιου λάθους σε οποιοδήποτε μέρος του κώδικα (είτε μέσα είτε έξω από το κρίσιμο τμήμα της), τότε η άλλη μπλοκάρεται επ' άπειρον.

30

Αλγόριθμος του Dekker: Δεύτερη προσπάθεια

```
enum boolean (false = 0; true = 1);
boolean flag[2] = (0, 0);

void P0()                              void P1()
{                                      {
 while (true) {                         while (true) {
    /* preceding code */ ;                  /* preceding code */ ;
    while (flag[1])                        while (flag[0])
       /* do nothing */ ;                      /* do nothing */ ;
    flag[0] = true;                        flag[1] = true;
    /* critical section */ ;               /* critical section */ ;
    flag[0] = false;                       flag[1] = false;
    /* following code */ ;                 /* following code */ ;
 }                                      }
}                                      }

void main()
{
 parbegin P0(); P1(); parend
}
```

31

31

Προβλήματα με τη δεύτερη προσέγγιση

- Το λάθος της πρώτης προσέγγισης ήταν ότι κρατούντο πληροφορίες μόνο για τη διεργασία που μπορούσε να εισέλθει στο κρίσιμο τμήμα, ενώ χρειάζονται πληροφορίες και για τις δύο διεργασίες.
- Αυτό επιτυγχάνεται με τη δεύτερη λύση (μέσω των μεταβλητών `flag`) και έτσι μία διεργασία και μπορεί να εισέλθει στο κρίσιμο τμήμα όσες φορές θέλει και δεν μπλοκάρεται αν ή άλλη διακόψει την εκτέλεσή της.
- Ας θεωρήσουμε όμως την ακόλουθη σειρά εκτέλεσης των εντολών των δύο διεργασιών:
  - Η `P0` εκτελεί το (δεύτερο) `while` και βρίσκει ότι η `flag[1]` είναι `false`.
  - Η `P1` εκτελεί το (δεύτερο) `while` και βρίσκει ότι η `flag[0]` είναι `false`.
  - Η `P0` θέτει `flag[0] = true` και μπαίνει στο κρίσιμο τμήμα.
  - Η `P1` θέτει `flag[1] = true` και μπαίνει στο κρίσιμο τμήμα.
- Δηλαδή αυτή η προσέγγιση δεν υποστηρίζει καν αμοιβαίο αποκλεισμό.

32

32

Αλγόριθμος του Dekker: Τρίτη προσπάθεια

```
enum boolean (false = 0; true = 1);
boolean flag[2] = (0, 0);

void P0()                              void P1()
{                                      {
 while (true) {                         while (true) {
    /* preceding code */ ;                  /* preceding code */ ;
    flag[0] = true;                        flag[1] = true;
    while (flag[1])                        while (flag[0])
       /* do nothing */ ;                      /* do nothing */ ;
    /* critical section */ ;               /* critical section */ ;
    flag[0] = false;                       flag[1] = false;
    /* following code */ ;                 /* following code */ ;
 }                                      }
}                                      }

void main()
{
 parbegin P0(); P1(); parend
}
```

33

33

Προβλήματα με την τρίτη προσέγγιση

- Το πρόβλημα με την δεύτερη προσέγγιση έγκειται στο ότι μία διεργασία μπορεί να αλλάξει την κατάσταση τής μεταβλητής της αφού η άλλη την έχει ελέγξει.
- Η τρίτη προσέγγιση προσπαθεί να λύσει το πρόβλημα που αναφέρθηκε προηγουμένως με το να υποχρεώνει κάθε διεργασία να δηλώσει πρώτα ότι επιθυμεί να εισέλθει στο κρίσιμο τμήμα της πριν αρχίσει να εκτελεί το βρόχο (ουσιαστικά εναλλάσσουμε τις δύο γραμμές κώδικα).
- Αν και πράγματι αυτή η λύση υποστηρίζει σωστά τον αμοιβαίο αποκλεισμό εν τούτοις εισαγάγει 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 και δεν θα εισέλθει ποτέ σε αυτό.

34

34

Αλγόριθμος του Dekker: Τέταρτη προσπάθεια

```
enum boolean (false = 0; true = 1);
boolean flag[2] = (0, 0);

void P0()                              void P1()
{                                      {
 while (true) {                         while (true) {
    /* preceding code */ ;                  /* preceding code */ ;
    flag[0] = true;                        flag[1] = true;
    while (flag[1]) {                      while (flag[0]) {
       flag[0] = false;                       flag[1] = false;
       delay(random_num)                      delay(random_num);
       flag[0] = true;                        flag[1] = true;
    }                                      }
    /* critical section */ ;               /* critical section */ ;
    flag[0] = false;                       flag[1] = false;
    /* following code */ ;                 /* following code */ ;
 }                                      }
}                                      }

void main()
{
 parbegin P0(); P1(); parend
}
```

35

35

Προβλήματα με την τέταρτη προσέγγιση

- Το πρόβλημα με την τρίτη προσέγγιση είναι ότι η κάθε διεργασία επιμένει να θέλει να εισέλθει στο κρίσιμο τμήμα και δεν έχει τη δυνατότητα να υποχωρήσει.
- 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 για ένα μικρό αλλά τυχαίο χρονικό διάστημα, αν το ίδιο επιθυμεί και η άλλη διεργασία.
- 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 `P0` θέτει `flag[0] = true`.
  - Η `P1` θέτει `flag[1] = true`.
  - Η `P0` ελέγχει την τιμή της `flag[1]`.
  - Η `P1` ελέγχει την τιμή της `flag[0]`.
  - Η `P0` θέτει `flag[0] = false`.
  - Η `P1` θέτει `flag[1] = false`.
  - Η `P0` θέτει `flag[0] = true`.
  - Η `P1` θέτει `flag[1] = true`.
- Αυτή η αλληλουχία εκτέλεσης εντολών μπορεί να συνεχίσει επ' άπειρον χωρίς καμία από τις δύο διεργασίες να μπορέσει ποτέ να εκτελέσει το κρίσιμο τμήμα της.
- Το φαινόμενο αυτό είναι γνωστό ως ενεργό αδιέξοδο (livelock).

36

36

Αλγόριθμος του Dekker: Πέμπτη προσπάθεια

```
enum boolean (false = 0; true = 1);
boolean flag[2] = (0, 0);
int turn;

void P0()                                   void P1()
{                                           {
 while (true) {                              while (true) {
    /* preceding code */ ;                       /* preceding code */ ;
    flag[0] = true;                             flag[1] = true;
    while (flag[1]) {                           while (flag[0]) {
       if (turn == 1) {                            if (turn == 0) {
          flag[0] = false;                            flag[1] = false;
          while (turn == 1)                           while (turn == 0)
             /* do nothing */ ;                          /* do nothing */ ;
          flag[0] = true;                             flag[1] = true;
       }                                           }
    }                                           }
    /* critical section */ ;                    /* critical section */ ;
    turn = 1;                                   turn = 0;
    flag[0] = false;                            flag[1] = false;
    /* following code */ ;                      /* following code */ ;
 }                                           }
}                                           }
void main ()
{
 flag [0] = false; flag [1] = false; turn = 0; parbegin P0(); P1(); parend
}
```

37

37

Αντιμετώπιση των προβλημάτων στην πέμπτη προσέγγιση

- Η τελευταία αυτή προσέγγιση επιλύει τα προηγούμενα προβλήματα με το να συνδυάζει τις τεχνικές των προηγούμενων προσεγγίσεων.
- Συγκεκριμένα, χρησιμοποιεί και τη μεταβλητή `turn` η οποία δηλώνει ποια διεργασία έχει προτεραιότητα να εισέλθει στο κρίσιμο τμήμα και τις μεταβλητές `flag` που χρησιμοποιούν οι διεργασίες για να δηλώσουν ότι θέλουν να έχουν πρόσβαση στο κρίσιμο τμήμα.
- Το πώς ακριβώς επιτυγχάνεται η σωστή υλοποίηση του αμοιβαίου αποκλεισμού, αφήνεται ως άσκηση.

38

38

Αλγόριθμος του Peterson

```
enum boolean (false = 0; true = 1);
boolean flag[2] = (0, 0);
int turn;

void P0()                                   void P1()
{                                           {
 while (true) {                              while (true) {
    /* preceding code */ ;                       /* preceding code */ ;
    flag[0] = true;                             flag[1] = true;
    turn = 1;                                   turn = 0;
    while (flag[1] && turn == 1)                while (flag[0] && turn == 0)
       /* do nothing */ ;                          /* do nothing */ ;
    /* critical section */ ;                    /* critical section */ ;
    flag[0] = false;                            flag[1] = false;
    /* following code */ ;                      /* following code */ ;
 }                                           }
}                                           }
void main ()
{
 flag [0] = false; flag [1] = false; parbegin P0(); P1(); parend
}
```

39

39

Σύγκριση των δύο αλγορίθμων

- Ο αλγόριθμος του Peterson (που δημοσιεύτηκε 16 χρόνια μετά τη δημοσίευση του αλγόριθμου του Dekker) είναι πιο απλός.
- Η βασική φιλοσοφία του αλγόριθμου είναι ότι ενώ μία διεργασία δηλώνει την πρόθεσή της να εισέλθει στο κρίσιμο τμήμα αφήνει την άλλη να πράξει αυτό αν το επιθυμεί.
- Το πώς ακριβώς επιτυγχάνεται η σωστή υλοποίηση του αμοιβαίου αποκλεισμού, αφήνεται και πάλι ως άσκηση.

40

40

Τα μειονεκτήματα της πρώτης κατηγορίας τεχνικών υλοποίησης του αμοιβαίου αποκλεισμού

- Όλοι οι αλγόριθμοι υλοποίησης του αμοιβαίου αποκλεισμού με προγραμματιστικές τεχνικές βασίζονται στη συνεχή εξέταση της τιμής μίας ή περισσοτέρων μεταβλητών.
- Αυτό επιτυγχάνεται με τη δημιουργία ατέρμονων βρόχων οι οποίοι εκτελούνται συνέχεια, σπαταλώντας το χρόνο της ΚΜΕ.
- Αυτό το φαινόμενο ονομάζεται ενεργός αναμονή (busy waiting) και είναι το πιο σοβαρό μειονέκτημα αυτής της κατηγορίας τεχνικών.
- Επίσης, οι αλγόριθμοι αυτοί δεν είναι πάντα εύκολο να επεκταθούν στη γενική περίπτωση Ν διεργασιών, δημιουργώντας πολύπλοκο κώδικα ο οποίος είναι δύσκολο να αποσφαλματωθεί.

41

41

Περιεχόμενα

- Ταυτοχρονισμός, συντρέχων προγραμματισμός και η έννοια του αμοιβαίου αποκλεισμού.
- Υλοποίηση αμοιβαίου αποκλεισμού:
  - Σε επίπεδο λογισμικού.
  - Σε επίπεδο υλικού.
  - Σε επίπεδο λειτουργικού συστήματος.
  - Σε επίπεδο γλωσσών προγραμματισμού.
- Κλασσικά προβλήματα ταυτοχρονισμού.

42

42

Απενεργοποίηση διακοπτών

- Ξεκινώντας από τη σκέψη ότι αυτό που διακόπτει τη λειτουργία μίας διεργασίας είναι οι διακόπτες (interrupts) θα μπορούσαμε να επιτύχουμε αμοιβαίο αποκλεισμό με την απενεργοποίησή τους πριν την είσοδο στο κρίσιμο τμήμα και την επανεργοποίησή τους όταν η διεργασία εξέλθει από το κρίσιμο τμήμα.
- Η τεχνική αυτή δεν λειτουργεί σε συστήματα με πολλούς επεξεργαστές.
- Επίσης έχει μεγάλο κόστος στην εφαρμογή της επειδή κατά το διάστημα που οι διακόπτες είναι απενεργοποιημένοι πολλές λειτουργίες του συστήματος δεν μπορούν να εκτελεσθούν.

43

43

Υλοποίηση αμοιβαίου αποκλεισμού με χρήση διακοπτών

```
while (1) {
   /* preceding code */ ;
   /* disable interrupts */ ;
   /* critical section */ ;
   /* enable interrupts */ ;
   /* following code */ ;
}
```

44

44

Ειδικές εντολές χαμηλού επιπέδου που υλοποιούνται στο επίπεδο της μηχανής

- Test&Set
- Compare&Swap
  - Επίσης γνωστή και ως «compare and exchange»
- Exchange

45

45

Test&Set

```
boolean testset (int i)
{
 if (i == 0) {
    i = 1;
    return true;
 }
 else {
      return false;
 }
}
```

- Αν η τιμή της παραμέτρου `i` είναι 0 το κάνει 1 και δηλώνει επιτυχή εκτέλεση και αν η τιμή της `i` είναι 1 απλά δηλώνει αποτυχία.
- Ο έλεγχος και αλλαγή της τιμής της `i` γίνονται σαν *αδιαίρετες* (indivisible) πράξεις.

46

46

Υλοποίηση του αμοιβαίου αποκλεισμού με την εντολή Test&Set

```
const int n = N; /* no. of processes */
int bolt;

void P(int i)
{
 while (1)
 {
  /* preceding code */ ;
  while (!testset(bolt))
     /* do nothing */ ;
  /* critical section */ ;
  bolt = 0;
  /* following code */ ;
 }
}

void main()
{
 bolt = 0;
 parbegin P(0); P(1); …; P(n); parend
}
```

- Μία κοινή μεταβλητή `bolt` χρησιμοποιείται με αρχική τιμή 0.
- Η μόνη διεργασία που μπορεί να εισέλθει στο κρίσιμο τμήμα είναι αυτή που βρίσκει τη `bolt` με τιμή 0 (και την κάνει 1).
- Όλες οι άλλες διεργασίες περιμένουν στο δεύτερο `while` έως ότου αυτή που βρίσκεται στο κρίσιμο τμήμα εξέλθει και αλλάξει την τιμή του `bolt` πάλι σε 0.
- Μόνο τότε κάποια από τις διεργασίες που περιμένει στο δεύτερο `while` θα μπορέσει να εισέλθει στο κρίσιμο τμήμα.

47

47

Compare&Swap

```
int compare_and_swap (
              int *word,
              int testval,
              int newval)
{
 int oldval;
 oldval = *word;
 if (oldval == testval)
    *word = newval;
 return oldval;
}
```

- Ελέγχει τη τιμή μίας θέσης μνήμης (`*word`) σε σχέση με την τιμή της παραμέτρου `testval`.
- Αν οι τιμές είναι οι ίδιες, τότε η θέση μνήμης παίρνει την τιμή της παραμέτρου `newval`, διαφορετικά δεν υπάρχει τροποποίηση.
- Η συνάρτηση επιστρέφει την παλιά τιμή της θέσης μνήμης, επομένως η θέση μνήμης έχει αλλάξει τιμή αν η τιμή που επιστρέφεται είναι η ίδια με την τιμή της `testval`.
- Οι πράξεις είναι και πάλι αδιαίρετες.

48

48

Υλοποίηση του αμοιβαίου αποκλεισμού με την εντολή Compare&Swap

```
const int n = N; /* no. of processes */
int bolt;

void P(int i)
{
 while (1)
 {
  /* preceding code */ ;
  while (compare_and_swap(bolt,0,1) == 1)
     /* do nothing */ ;
  /* critical section */ ;
  bolt = 0;
  /* following code */ ;
 }
}

void main()
{
 bolt = 0;
 parbegin P(0); P(1); …; P(n); parend
}
```

- Μία κοινή μεταβλητή `bolt` χρησιμοποιείται με αρχική τιμή 0.
- Η μόνη διεργασία που μπορεί να εισέλθει στο κρίσιμο τμήμα είναι αυτή που βρίσκει τη `bolt` με τιμή 0 (και την κάνει 1).
- Όλες οι άλλες διεργασίες περιμένουν στο δεύτερο `while` έως ότου αυτή που βρίσκεται στο κρίσιμο τμήμα εξέλθει και αλλάξει την τιμή του `bolt` πάλι σε 0.
- Μόνο τότε κάποια από τις διεργασίες που περιμένει στο δεύτερο `while` θα μπορέσει να εισέλθει στο κρίσιμο τμήμα.

49

49

Exchange

```
void exchange (
     int register,
     int memory)
{
   int temp;
   temp = memory;
   memory = register;
   register = temp;
}
```

- Ανταλλάσσει τα περιεχόμενα του καταχωρητή `register` με αυτά της θέσης μνήμης `memory`.
- Όσο διαρκεί η εκτέλεση της εντολής, η θέση μνήμης είναι κλειδωμένη και δεν μπορεί να προσπελασθεί από κανέναν άλλο.
- Οι αρχιτεκτονικές Intel IA-32 (Pentium) και IA-64 (Itanium) υποστηρίζουν μία τέτοια εντολή (XCHG).

50

50

Υλοποίηση του αμοιβαίου αποκλεισμού με την εντολή Exchange

```
const int n = N; /* no. of processes */
int bolt;

void P(int i)
{
 int keyi;
 while (1)
 {
  /* preceding code */ ;
  keyi = 1;
  while (keyi != 0)
     exchange(keyi,bolt);
  /* critical section */ ;
  exchange(keyi,bolt);
  /* following code */ ;
 }
}

void main()
{
 bolt = 0;
 parbegin P(0); P(1); …; P(n); parend
}
```

- Μία κοινή μεταβλητή `bolt` χρησιμοποιείται με αρχική τιμή 0.
- Κάθε διεργασία έχει μία τοπική μεταβλητή `keyi` με αρχική τιμή 1.
- Η μόνη διεργασία που μπορεί να εισέλθει στο κρίσιμο τμήμα είναι αυτή που βρίσκει τη `bolt` με τιμή 0 (και την κάνει 1).
- Όλες οι άλλες διεργασίες περιμένουν στο δεύτερο `while` έως ότου αυτή που βρίσκεται στο κρίσιμο τμήμα εξέλθει και αλλάξει την τιμή του `bolt` πάλι σε 0.
- Μόνο τότε κάποια από τις διεργασίες που περιμένει στο δεύτερο `while` θα μπορέσει να εισέλθει στο κρίσιμο τμήμα.

51

51

Πλεονεκτήματα της υλοποίησης του αμοιβαίου αποκλεισμού σε επίπεδο υλικού

- 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.

52

52

Μειονεκτήματα της υλοποίησης του αμοιβαίου αποκλεισμού σε επίπεδο υλικού

- Γίνεται χρήση ενεργούς αναμονής με αρνητικό αποτέλεσμα το σχετικό κόστος.
- 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 αδιέξοδου στην περίπτωση που μία διεργασία ενώ βρίσκεται στο κρίσιμο τμήμα της διακοπεί από μία άλλη διεργασία μεγαλύτερης προτεραιότητας. Π.χ. αν σε ένα σύστημα με ένα επεξεργαστή μία διεργασία `P0` ενώ βρίσκεται σε ένα κρίσιμο τμήμα διακοπεί από μία άλλη διεργασία `P1` η οποία έχει μεγαλύτερη προτεραιότητα από την `P0`, τότε αν επιπλέον η `P1` προσπαθήσει να εισέλθει στο ίδιο κρίσιμο τμήμα θα έχουμε αδιέξοδο: η `P1` δεν θα μπορεί να συνεχίσει αλλά λόγω του ότι έχει μεγαλύτερη προτεραιότητα δεν θα αφήσει και την `P0` να συνεχίσει.

53

53

Περιεχόμενα

- Ταυτοχρονισμός, συντρέχων προγραμματισμός και η έννοια του αμοιβαίου αποκλεισμού.
- Υλοποίηση αμοιβαίου αποκλεισμού:
  - Σε επίπεδο λογισμικού.
  - Σε επίπεδο υλικού.
  - → Σε επίπεδο λειτουργικού συστήματος.
  - Σε επίπεδο γλωσσών προγραμματισμού.
- Κλασσικά προβλήματα ταυτοχρονισμού.

54

54

Σημαφόρος

- Η βασική αρχή στηρίζεται στη χρήση σημάτων (signals): δύο ή περισσότερες διεργασίες μπορούν να συγχρονίσουν την εκτέλεσή τους μέσω της αποστολής και αναμονής λήψης σημάτων.
- Για την αποστολή και καταμέτρηση των σημάτων χρησιμοποιούνται ειδικές μεταβλητές που λέγονται σημαφόροι (semaphores) σε συνδυασμό με δύο θεμελιώδεις λειτουργίες: `semWait` και `semSignal`.
- Οι σημαφόροι προτάθηκαν από τον Dijkstra το 1965.

55

55

Πράξεις με σημαφόρους

- Μόνο 3 πράξεις μπορούν να γίνουν με ένα σημαφόρο:
  - Αρχικοποίηση με μία ακέραια μη αρνητική τιμή.
  - Μείωση της τιμής του σημαφόρου κατά μία μονάδα με χρήση της εντολής `semWait`. Αν η τιμή του σημαφόρου γίνει αρνητική, η διεργασία που εκτέλεσε την εντολή αναστέλλει την εκτέλεσή της και μπαίνει σε μία ουρά υπό αναστολή διεργασιών σχετιζόμενη με το σημαφόρο. Διαφορετικά, η διεργασία συνεχίζει την εκτέλεσή της.
  - Αύξηση της τιμής του σημαφόρου κατά μία μονάδα με χρήση της εντολής `semSignal`. Αν η τιμή του σημαφόρου δεν είναι θετική, τότε μία από τις διεργασίες που βρίσκονται υπό αναστολή στην ουρά του σημαφόρου (αν υπάρχουν τέτοιες διεργασίες) αλλάζει την κατάστασή της σε έτοιμη για εκτέλεση.
  - Εξυπακούεται ότι η εκτέλεση των εντολών `semWait` και `semSignal` θα πρέπει να γίνεται ως *ατομική ενέργεια* (atomic action).

56

56

Ορισμός σημαφόρου

```
struct semaphore {
   int count;
   queueType queue;
};

void semWait(semaphore s)
{
 s.count--;
 if (s.count < 0) {
    /* place this process in s.queue */ ;
    /* block this process */ ;
 }
}

void semSignal(semaphore s)
{
 s.count++;
 if (s.count <= 0) {
    /* remove a process P from s.queue */ ;
    /* place process P on ready list */ ;
 }
}
```

57

57

Δυαδικός σημαφόρος

- 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.
- Μία παρόμοια έννοια είναι αυτή του μηχανισμού `mutex`. Η βασική διαφορά από τον δυαδικό σημαφόρο είναι ότι η διεργασία που κλειδώνει το `mutex` (θέτει την τιμή του σε 0) είναι η μόνη που μπορεί να το ξεκλειδώσει (να θέσει την τιμή του σε 1). Στους δυαδικούς σημαφόρους δεν ισχύει αυτός ο περιορισμός και η διεργασία που ξεκλειδώνει ένα δυαδικό σημαφόρο δεν είναι κατ' ανάγκη η ίδια που τον κλείδωσε.

58

58

Πράξεις με δυαδικούς σημαφόρους

- Όπως και προηγουμένως, μόνο 3 πράξεις μπορούν να γίνουν με ένα δυαδικό σημαφόρο:
  - Αρχικοποίηση σε 0 ή 1.
  - Η εντολή `semWaitB` ελέγχει την τιμή του σημαφόρου. Αν αυτή είναι ήδη 0, η διεργασία που εκτέλεσε την εντολή αναστέλλει την εκτέλεσή της και μπαίνει σε μία ουρά υπό αναστολή διεργασιών σχετιζόμενη με το σημαφόρο. Διαφορετικά, αν η τιμή του σημαφόρου είναι 1 τότε γίνεται 0 και η διεργασία συνεχίζει την εκτέλεσή της.
  - Η εντολή `semSignalB` ελέγχει αν υπάρχουν διεργασίες υπό αναστολή στην ουρά του σημαφόρου (με τιμή σημαφόρου 0). Αν υπάρχουν, τότε μία από αυτές αλλάζει την κατάστασή της σε έτοιμη για εκτέλεση. Αν η ουρά είναι άδεια, τότε η τιμή του σημαφόρου γίνεται 1. Αν η τιμή του σημαφόρου είναι ήδη 1 και η ουρά του άδεια, η εκτέλεση της `semSignalB` δεν έχει κανένα αποτέλεσμα.
  - Και πάλι, η εκτέλεση των εντολών `semWaitB` και `semSignalB` θα πρέπει να γίνεται ως ατομική ενέργεια.

59

59

Ορισμός δυαδικού σημαφόρου

```
struct binary_semaphore {
   enum {zero, one} value;
   queueType queue;
};

void semWaitB(binary_semaphore s)
{
 if (s.value == one)
    s.value = zero;
 else {
       /* place this process in s.queue */ ;
       /* block this process */ ;
 }
}

void semSignalB(binary_semaphore s)
{
 if (s.queue is empty())
    s.value = one;
 else {
       /* remove a process P from s.queue */ ;
       /* place process P on ready list */ ;
 }
}
```

60

60

Ισχυρός/Αδύνατος σημαφόρος

- 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 (strong semaphore).
- Στην αντίθετη περίπτωση που δεν καθορίζεται η σειρά ενεργοποίησης των διεργασιών, ο σημαφόρος αυτός λέγεται αδύνατος (weak semaphore).

61

Παράδειγμα λειτουργίας ενός ισχυρού σημαφόρου — 1

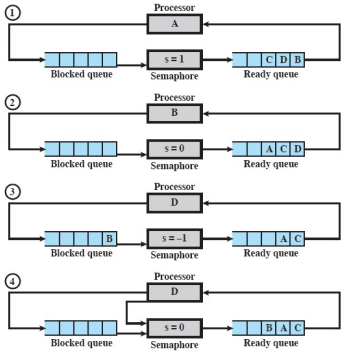
```
sema s=0;
int result;

void A()          void B()          void C()          void D()
{                 {                 {                 {
 while (1)         while (1)         while (1)         while(1)
 {                 {                 {                 {
  semWait(s);       semWait(s);       semWait(s);       produce(result);
  use(result);      use(result);      use(result);      semSignal(s);
 }                 }                 }                 }
}                 }                 }                 }

parbegin
   A(); B(); C(); D();
parend
```


62

Παράδειγμα λειτουργίας ενός ισχυρού σημαφόρου — 2



63

Παράδειγμα λειτουργίας ενός ισχυρού σημαφόρου — 3

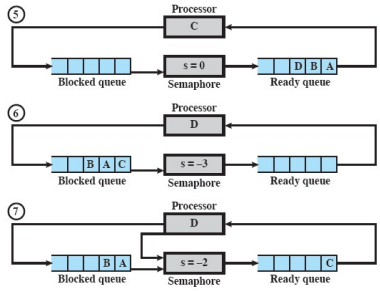


Figure 5.5 Example of Semaphore Mechanism

64

Υλοποίηση σημαφόρων

- Όπως ήδη επιπώθηκε, είναι πολύ σημαντικό η εκτέλεση των εντολών `semaWait` και `semaSignal` (ή `semaWaitB` και `semaSignalB` για δυαδικούς σημαφόρους) να γίνεται ως ατομική πράξη.
- Μία ευνόητη λύση είναι η υλοποίησή τους στο υλικό ή ως μικροπρόγραμμα.
- Αν αυτό δεν είναι δυνατόν, μπορούν να χρησιμοποιηθούν κάποιες από τις προαναφερθείσες προσεγγίσεις υλοποίησης του αμοιβαίου αποκλεισμού, δεδομένου ότι η ουσία του προβλήματος είναι ακριβώς αυτή: μόνο μία διεργασία μπορεί ανά πάσα χρονική στιγμή να έχει πρόσβαση σε ένα σημαφόρο μέσω εκτέλεσης των εντολών του.
- Μπορούν επομένως να χρησιμοποιηθούν τεχνικές τύπου Dekker και Peterson, αν και το κόστος θα είναι σημαντικό.
- Εναλλακτικά, μπορούν να χρησιμοποιηθούν οι εντολές που εξετάστηκαν στο επίπεδο του υλικού ή η τεχνική της απενεργοποίησης διακοπτών.
- Αν και εδώ έχουμε χρήση ενεργούς αναμονής, το κόστος είναι περιορισμένο λόγω του ότι το μέγεθος του κρίσιμου τμήματος (πρόσβαση σε μία ακέραια μεταβλητή και μία ουρά και εκτέλεση λίγων απλών εντολών) είναι πολύ μικρό.

65

Υλοποίηση σημαφόρων με την εντολή Compare&Swap

```
semWait(s)
{
 while (compare_and_swap(s.flag, 0 , 1) == 1)
    /* do nothing */ ;
 s.count--;
 if (s.count < 0) {
    /* place this process in s.queue */ ;
    /* block this process (must also set s.flag to 0) */ ;
 }
 s.flag = 0;
}

semSignal(s)
{
 while (compare_and_swap(s.flag, 0 , 1) == 1)
    /* do nothing */ ;
 s.count++;
 if (s.count <= 0) {
    /* remove a process P from s.queue */ ;
    /* place process P on ready list */ ;
 }
 s.flag = 0;
}
```


66

Υλοποίηση σημαφόρων μέσω απενεργοποίησης των διακοπτών

```
semWait(s)
{
 inhibit interrupts;
 s.count--;
 if (s.count < 0) {
    /* place this process in s.queue */ ;
    /* block this process and allow interrupts */ ;
 }
 else
      allow interrupts;
}

semSignal(s)
{
 inhibit interrupts;
 s.count++;
 if (s.count <= 0) {
    /* remove a process P from s.queue */ ;
    /* place process P on ready list */ ;
 }
 allow interrupts;
}
```


67

Υλοποίηση του αμοιβαίου αποκλεισμού με σημαφόρους

```
const int n = N; /* number of processes */
semaphore s = 1;

void P(int i)
{
 while (1)
 {
  /* preceding code */ ;
  semWait(s);
  /* critical section */ ;
  semSignal(s);
  /* following code */ ;
 }
}

void main()
{
 parbegin P(0); P(1); …; P(n); parend
}
```


68

Σενάριο πρόσβασης σε κοινό πόρο από 3 διεργασίες μέσω σημαφόρου

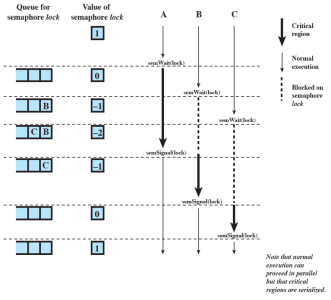


Figure 5.10 Processes Accessing Shared Data Protected by a Semaphore

69

Συμπεριφορά των διεργασιών στο προηγούμενο σενάριο

- Ο κοινός πόρος προστατεύεται από το σημαφόρο `lock`.
- Η διεργασία `A` εκτελεί την εντολή `semWait(lock)`.
  - Επειδή `lock==1`, η `A` εισέρχεται στο κρίσιμο τμήμα της και `lock=0`.
- Όσο η `A` βρίσκεται μέσα στο κρίσιμο τμήμα της, οι διεργασίες `B` και `C` εκτελούν την εντολή `semWait(lock)` και τίθενται υπό αναστολή.
- Όταν η `A` εξέλθει από το κρίσιμο τμήμα της και εκτελέσει την εντολή `semSignal(lock)`, η `B` που ήταν η πρώτη από τις άλλες δύο διεργασίες που τέθηκε υπό αναστολή (επομένως είναι η πρώτη στην ουρά του `lock`) μπορεί τώρα να εισέλθει στο κρίσιμο τμήμα της.

70

Πλεονεκτήματα των σημαφόρων έναντι προηγούμενων τεχνικών

- Ελαχιστοποίηση του κόστους που συνεπάγεται η ενεργός αναμονή˙ οι διεργασίες που δεν μπορούν να εισέλθουν στο κρίσιμο τμήμα τίθενται υπό αναστολή και δεν σπαταλούν πόρους του συστήματος.
- Η απρόσμενη διακοπή μίας διεργασίας λόγω κάποιου λάθους, ακόμα και αν γίνει μέσα στο κρίσιμο τμήμα, συνήθως δεν έχει καταστροφικά αποτελέσματα. Το Λ.Σ., μέσω εξέτασης των δομών των σημαφόρων που εμπλέκονται σε ένα σενάριο ταυτοχρονισμού, έχει συνολική εικόνα της κατάστασης και μπορεί να αντιδράσει (είτε τερματίζοντας όλες τις διεργασίες που εμπλέκονται στο σενάριο, είτε επιτρέποντας σε κάποιες από αυτές να συνεχίσουν την εκτέλεσή τους).
- Οι τεχνικές υλοποίησης του αμοιβαίου αποκλεισμού είναι εύκολα επεκτάσιμες σε σενάρια που εμπλέκουν πολλαπλές διεργασίες.
- Ο κώδικας συνολικά είναι πιο απλός και κατανοητός.

71

Μειονεκτήματα των σημαφόρων

- Ο προγραμματιστής συνεχίζει να είναι υπεύθυνος και για την υλοποίηση του αμοιβαίου αποκλεισμού και για το συγχρονισμό μεταξύ των διεργασιών.
- Ειδικά σε πολύπλοκα σενάρια ταυτοχρονισμού όπου γίνεται χρήση πολλών σημαφόρων, μπορεί εύκολα να δημιουργηθούν λάθη στο ζευγάρωμα των εντολών `semWait` και `semSignal`, προκαλώντας έτσι αδιέξοδο, παρατεταμένη στέρηση ή μη αποτελεσματική υλοποίηση του αμοιβαίου αποκλεισμού (δες και παραδείγματα αργότερα).

72

Μηνύματα

- Η διαδραστική σχέση μεταξύ των διεργασιών οδηγεί στην ανάγκη υποστήριξης:
  - Συγχρονισμού (μεταξύ άλλων και για υλοποίηση αμοιβαίου αποκλεισμού).
  - Επικοινωνίας (για ανταλλαγή πληροφοριών μεταξύ συνεργαζόμενων διεργασιών).
- Η δεύτερη ανάγκη υποστηρίζεται με τη χρήση μηνυμάτων (messages).
- Ο μηχανισμός των μηνυμάτων είναι αρκετά ευέλικτος για να μπορεί να υλοποιηθεί σε οποιοδήποτε σύστημα, με κοινή ή κατανεμημένη μνήμη και με έναν ή περισσότερους επεξεργαστές.

73

73

Ανταλλαγή μηνυμάτων

- Αν και υπάρχουν πολλοί επί μέρους μηχανισμοί για ανταλλαγή μηνυμάτων μεταξύ διεργασιών, στη γενική περίπτωση υποστηρίζονται δύο είδη θεμελιωδών εντολών:
  - `send(`διεύθυνση *παραλήπτη*`,` μήνυμα`)`, όπου η διεργασία που εκτελεί την εντολή αυτή στέλνει κάποιο μήνυμα σε μία άλλη διεργασία (παραλήπτης).
  - `receive(`διεύθυνση *αποστολέα*`,` μήνυμα`)`, όπου η διεργασία που εκτελεί την εντολή αυτή παραλαμβάνει κάποιο μήνυμα από μία άλλη διεργασία (αποστολέας).

74

74

Συγχρονισμός μεταξύ των διεργασιών με χρήση μηνυμάτων

- Η επικοινωνία μεταξύ των διεργασιών προϋποθέτει κάποιο είδος συγχρονισμού μεταξύ τους.
  - Ο παραλήπτης δεν μπορεί να παραλάβει κάποιο μήνυμα αν δεν το στείλει πρώτα ο αποστολέας.
- Όταν μία διεργασία εκτελέσει την εντολή `send`, υπάρχουν δύο πιθανότητες:
  - Η διεργασία τίθεται υπό αναστολή μέχρις ότου το μήνυμα φτάσει στον παραλήπτη.
  - Η διεργασία συνεχίζει κανονικά την εκτέλεσή της.
- Όταν μία διεργασία εκτελέσει την εντολή `receive`, υπάρχουν και πάλι δύο πιθανότητες:
  - Αν το μήνυμα έχει ήδη φτάσει, η διεργασία το χρησιμοποιεί και συνεχίζει την εκτέλεση της.
  - Αν το μήνυμα δεν έχει φτάσει, τότε η διεργασία:
    - Είτε τίθεται υπό αναστολή μέχρις ότου φτάσει το μήνυμα.
    - Είτε εγκαταλείπει την προσπάθειά της να πάρει το μήνυμα και συνεχίζει την εκτέλεσή της.

75

75

Όταν και οι δύο εντολές θέτουν διεργασίες υπό αναστολή

- Σε αυτό το μοντέλο και ο αποστολέας και ο παραλήπτης τίθενται υπό αναστολή μέχρις ότου το μήνυμα φτάσει στον παραλήπτη (blocking `send`, blocking `receive`).
- Γνωστό και ως ραντεβού (rendezvous).
- Επιτρέπει αυστηρό συγχρονισμό μεταξύ των διεργασιών.
- Χρησιμοποιήθηκε στη γλώσσα προγραμματισμού Ada.

76

76

Όταν η εντολή `send` δεν θέτει την διεργασία που την εκτέλεσε υπό αναστολή

- Πιο φυσικός τρόπος επικοινωνίας μεταξύ των διεργασιών (non blocking `send`, blocking `receive`).
- Ο αποστολέας συνεχίζει την εκτέλεση των εντολών μετά την εκτέλεση της εντολής `send` χωρίς να περιμένει το μήνυμα να φτάσει στον παραλήπτη.
- Ο παραλήπτης όμως τίθεται υπό αναστολή μέχρις ότι παραλάβει το μήνυμα.
- Υπάρχει και η παραλλαγή όπου ο παραλήπτης δεν υποχρεούται να περιμένει την έλευση του μηνύματος (non blocking `send`, non blocking `receive`).

77

77

Καθορισμός διευθύνσεων στα μηνύματα

- Υπάρχει ανάγκη να καθορισθεί η διεύθυνση του παραλήπτη σε μία εντολή τύπου `send` και του αποστολέα σε μία εντολή τύπου `receive`.
- Υπάρχουν δύο τρόποι καθορισμού της διεύθυνσης ενός αποστολέα ή παραλήπτη:
  - Άμεση.
  - Έμμεση.

78

78

Άμεσος καθορισμός διεύθυνσης

- Η εντολή `send` περιλαμβάνει τη διεύθυνση μνήμης μίας συγκεκριμένης διεργασίας που είναι ο παραλήπτης του μηνύματος.
- Αντίστοιχα η εντολή `receive` είτε καθορίζει και αυτή τη διεύθυνση μνήμης ενός συγκεκριμένου αποστολέα είτε αν ο αποστολέας δεν είναι γνωστός η σχετική παράμετρος παίρνει τιμή με την παραλαβή του μηνύματος.

79

Έμμεσος καθορισμός διεύθυνσης

- Τα μηνύματα στέλνονται σε μία κοινή δομή δεδομένων που αποτελείται από ουρές.
- Αυτές οι ουρές ονομάζονται γραμματοκιβώτια (mailboxes).
- Ο αποστολέας στέλνει το μήνυμα σε ένα γραμματοκιβώτιο και ο παραλήπτης παίρνει το μήνυμα από αυτό το γραμματοκιβώτιο.

80

Έμμεση επικοινωνία μεταξύ των διερνασιών — 1

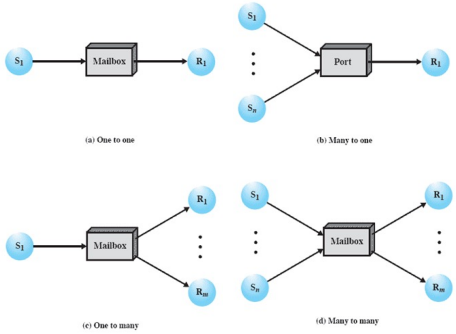


Figure 5.18 Indirect Process Communication

81

Έμμεση επικοινωνία μεταξύ των διεργασιών — 2

- Σενάριο (α), σχέση μία-προς-μία: επιτρέπει τη δημιουργία ενός προσωπικού καναλιού επικοινωνίας μεταξύ δύο διεργασιών, χωρίς να είναι δυνατή η παρεμβολή άλλων.
- Σενάριο (β), σχέση πολλές-προς-μία: εδώ μία διεργασία προσφέρει υπηρεσίες σε ένα αριθμό από άλλες διεργασίες. Το μοντέλο αυτό είναι χρήσιμο σε περιβάλλοντα εξυπηρετητή-πελάτη (client-server) και σε αυτήν την περίπτωση το γραμματοκιβώτιο συνήθως λέγεται θύρα (port).
- Σενάριο (γ), σχέση μία-προς-πολλές: επιτρέπει την εκπομπή πληροφοριών από μία διεργασία σε άλλες.
- Σενάριο (δ), σχέση πολλές-προς-πολλές: επιτρέπει σε πολλαπλές εξυπηρετητές διεργασίες να επικοινωνούν με πολλαπλές πελάτες διεργασίες για να προσφέρουν ταυτόχρονες υπηρεσίες.

82

Γενική δομή ενός μηνύματος

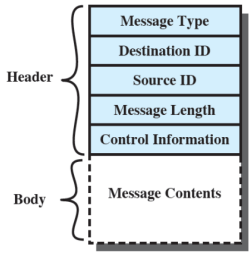


Figure 5.22 General Message Format

83

Υλοποίηση του αμοιβαίου αποκλεισμού με χρήση μηνυμάτων

```
const int n = N; /* number of processes */

void P(int i)
{
 message msg;
 while (1)
 {
  /* preceding code */ ;
  receive (box, msg); /* receive is blocking */
  /* critical section */ ;
  send (box, msg); /* send is non blocking */
  /* remaining code */;
 }
}

void main()
{
 create_mailbox (box);
 send (box, null);
 parbegin P(0); P(1); …; P(n); parend
}
```


84

Περιεχόμενα

– Ταυτοχρονισμός, συντρέχων προγραμματισμός και η έννοια του αμοιβαίου αποκλεισμού.

– Υλοποίηση αμοιβαίου αποκλεισμού:
- Σε επίπεδο λογισμικού.
- Σε επίπεδο υλικού.
- Σε επίπεδο λειτουργικού συστήματος.
- Σε επίπεδο γλωσσών προγραμματισμού.

– Κλασσικά προβλήματα ταυτοχρονισμού.

Παρακολουθητής

- Ο παρακολουθητής (monitor) είναι μία δομή που παρέχει λειτουργικότητα ισοδύναμη με αυτή του σημαφόρου αλλά είναι πιο εύκολη η χρήση του.
- Προτάθηκε από τους Tony Hoare και Brinch Hansen το 1974-75.
- Μερικές από τις γλώσσες προγραμματισμού που υποστηρίζουν παρακολουθητές είναι οι εξής:
  - Concurrent Pascal, Pascal-Plus,
  - Modula-2, Modula-3, και Java.

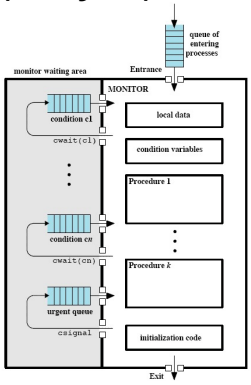
Βασικά χαρακτηριστικά του παρακολουθητή

- Οι τοπικές δομές δεδομένων και μεταβλητές μέσα σε έναν παρακολουθητή είναι προσβάσιμες μόνο από τις συναρτήσεις του παρακολουθητή και όχι από εξωτερικές συναρτήσεις.
- Μία διεργασία εισέρχεται σε έναν παρακολουθητή μόνο μέσω της κλήσης κάποιας από τις συναρτήσεις του παρακολουθητή (καθιστώντας έτσι τον παρακολουθητή προπομπό του αντικειμενοστρεφούς προγραμματισμού).
- Μόνο μία διεργασία ανά πάσα χρονική στιγμή μπορεί να βρίσκεται μέσα σε έναν παρακολουθητή˙ οποιαδήποτε άλλη διεργασία που προσπαθεί να εισέλθει στον παρακολουθητή τίθεται υπό αναστολή μέχρις ότου αυτός είναι και πάλι διαθέσιμος.

Συγχρονισμός στη χρήση ενός παρακολουθητή

- Επιτυγχάνεται με τη χρήση μεταβλητών συνθήκης (condition variables), που βρίσκονται μέσα στον παρακολουθητή και είναι προσβάσιμες μόνο από αυτόν.
- Η χρήση τους γίνεται μέσω δυο εντολών:
  - `cwait(c)`: θέτει υπό αναστολή τη διεργασία που την εκτελεί στην ουρά της μεταβλητής συνθήκης `c` (ο παρακολουθητής είναι τώρα διαθέσιμος στις υπόλοιπες διεργασίες).
  - `csignal(c)`: ενεργοποιεί τυχαία κάποια από τις διεργασίες που βρίσκονται υπό αναστολή στην ουρά της μεταβλητής συνθήκης `c` (αν η ουρά είναι άδεια τότε η εκτέλεση της εντολής είναι σαν να μην έγινε)˙ ο παρακολουθητής τώρα χρησιμοποιείται (αποκλειστικά) από τη διεργασία που ενεργοποιήθηκε.
- Σημειωτέον, ότι η εντολή `csignal` λειτουργεί διαφορετικά από την εντολή `semSignal` ενός (γενικού) σημαφόρου. Στη δεύτερη περίπτωση ακόμα και αν δεν υπάρχει καμία διεργασία υπό αναστολή αναφορικά με κάποιον σημαφόρο, εκτελώντας ένα `semSignal` για αυτόν οδηγεί στην αύξηση της τιμής του κατά μία μονάδα. Στην πρώτη περίπτωση η εκτέλεση της `csignal` για κάποια μεταβλητή συνθήκης απλά αγνοείται αν δεν υπάρχει καμία διεργασία υπό αναστολή αναφορικά με τη συνθήκη αυτή.

Δομή ενός παρακολουθητή



Επεξήγηση της δομής ενός παρακολουθητή

- Αν και η πρόσβαση στον παρακολουθητή από μία διεργασία γίνεται μέσω της κλήσης οποιασδήποτε συνάρτησης του παρακολουθητή, μπορούμε να φανταστούμε ότι υπάρχει ένα μόνο σημείο πρόσβασης από το οποίο εισέρχονται οι διεργασίες στον παρακολουθητή.
- Μόνο μία διεργασία ανά πάσα στιγμή μπορεί να είναι μέσα στον παρακολουθητή και οι υπόλοιπες περιμένουν από έξω σε μία ουρά.
- Η διεργασία που βρίσκεται μέσα στον παρακολουθητή μπορεί να θέσει τον εαυτό της υπό αναστολή στην ουρά κάποιας μεταβλητής συνθήκης `c` με την εκτέλεση της εντολής `cwait(c)` και θα παραμείνει εκεί μέχρις ότου κάποια άλλη διεργασία την ενεργοποιήσει μέσω της εντολής `csignal(c)` οπότε και θα συνεχίσει την εκτέλεσή της με τις εντολές που βρίσκονται μετά την εντολή `cwait(c)`.
- Αν η διεργασία που βρίσκεται στον παρακολουθητή διαπιστώσει κάποια αλλαγή στην κατάσταση της συνθήκης `c`, εκτελεί την εντολή `csignal(c)` οπότε και ενεργοποιεί κάποια από τις διεργασίες (αν υπάρχουν) που βρίσκονται υπό αναστολή στην ουρά της `c`.

Υλοποίηση του αμοιβαίου αποκλεισμού με χρήση παρακολουθητή

```
monitor mutual_exclusion()
{
 boolean in_use = false;
 cond avail;

 void enter_critical()
 {
  if (in_use)
     cwait(avail);
  in_use = true;
 }

 void exit_critical()
 {
  in_use = false;
  csignal(avail);
 }
}
```

```
const int n = N;

void P(int i)
{
 while (1)
 {
  /* preceding code */
  mutual_exclusion.enter_critical();
  /* critical section */ ;
  mutual_exclusion.exit_critical();
  /* following code */ ;
 }
}

void main()
{
 parbegin
    P(0); P(1); …; P(n);
 parend
}
```


Πλεονεκτήματα των παρακολουθητών έναντι των σημαφόρων

- Η ίδια η φύση του παρακολουθητή επιβάλει την εφαρμογή του αμοιβαίου αποκλεισμού. Ο προγραμματιστής είναι υπεύθυνος μόνο για τον συγχρονισμό των διεργασιών.
- Αυτός ο συγχρονισμός επιτυγχάνεται με τη χρήση των `cwait` και `csignal` που πρέπει να γίνει σωστά. Όμως, όταν ελεγχτεί και απασφαλματωθεί ο κώδικας του παρακολουθητή, μπορεί μετά να χρησιμοποιηθεί από τον οποιοδήποτε, όπως γίνεται με κάποια συνάρτηση βιβλιοθήκης.
- Κατ' επέκταση, όταν ο κώδικας του παρακολουθητή δεν έχει λάθη, είναι εγγυημένο ότι οι διεργασίες επίσης δεν έχουν λάθη (σε σχέση με την υλοποίηση του αμοιβαίου αποκλεισμού). Στην περίπτωση των σημαφόρων, πρέπει να ελέγχονται όλες οι διεργασίες.

Μειονεκτήματα του παρακολουθητή

- Ο ορισμός του παρακολουθητή, όπως διαμορφώθηκε από τους Hoare και Hansen, επιβάλλει ότι όταν εκτελείται μία εντολή `csignal(c)` θα πρέπει *αμέσως* να ενεργοποιηθεί και να αρχίσει εκτέλεση κάποια από τις διεργασίες που είναι υπό αναστολή στη συνθήκη `c`.
- Επομένως, η διεργασία που εκτέλεσε τη `csignal` θα πρέπει ή να τερματίσει αμέσως την εκτέλεσή της ή να την αναστείλει. Αυτό δημιουργεί ορισμένα προβλήματα:
  - Αν η διεργασία που εκτέλεσε τη `csignal` δεν έχει ολοκληρώσει την εκτέλεσή της, τότε θα χρειασθούν δύο επιπλέον εναλλαγές περιβάλλοντος για αυτήν (αναστολή και επανεργοποίηση). Εναλλακτικά, θα πρέπει να απαγορευθεί συντακτικά να υπάρχουν άλλες εντολές στις συνάρτησεις μετά από μία `csignal` που ουσιαστικά θα πρέπει να είναι η τελευταία εντολή της συνάρτησης (αυτός ο περιορισμός υιοθετήθηκε στη γλώσσα Concurrent Pascal).
  - Ο μηχανισμός εναλλαγής στην ΚΜΕ της διεργασίας που εκτέλεσε την `csignal(c)` με κάποια άλλη διεργασία υπό αναστολή στη `c`, πρέπει να γίνει με ακρίβεια διαφορετικά κάποια τρίτη διεργασία μπορεί να προλάβει να εκτελεσθεί πρώτη και να αλλάξει την `c`.

Εναλλακτικός ορισμός του παρακολουθητή

- Οι Lampson και Redell πρότειναν ένα διαφορετικό ορισμό του παρακολουθητή όπου αντί για τη `csignal` υπάρχει η εντολή `cnotify`. Η παραλλαγή αυτή χρησιμοποιήθηκε στο Λ.Σ. Mesa.
- Ο ορισμός της `cnotify` είναι ο εξής: η εκτέλεση της εντολής `cnotify(c)` ναι μεν μεταφέρει το μήνυμα στην ουρά των υπό αναστολή διεργασιών στη συνθήκη `c` ότι κάποια από αυτές θα πρέπει κάποια στιγμή να ενεργοποιηθεί, αλλά συνεχίζει την εκτέλεση της διεργασίας που έκανε χρήση της `cnotify`. 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 τη `cnotify` 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.

Υλοποίηση του αμοιβαίου αποκλεισμού με χρήση παρακολουθητή (με `cnotify`)

```
monitor mutual_exclusion()
{
 boolean in_use = false;
 cond avail;

 void enter_critical()
 {
  while (in_use)
     cwait(avail);
  in_use = true;
 }

 void exit_critical()
 {
  in_use = false;
  cnotify(avail);
 }
}
```

```
const int n = N;

void P(int i)
{
 while (1)
 {
  /* preceding code */
  mutual_exclusion.enter_critical().
  /* critical section */ ;
  mutual_exclusion.exit_critical().
  /* following code */ ;
 }
}

void main()
{
 parbegin
    P(0); P(1); …; P(n);
 parend
}
```


Περιεχόμενα

- Ταυτοχρονισμός, συντρέχων προγραμματισμός και η έννοια του αμοιβαίου αποκλεισμού.
- Υλοποίηση αμοιβαίου αποκλεισμού:
  - Σε επίπεδο λογισμικού.
  - Σε επίπεδο υλικού.
  - Σε επίπεδο λειτουργικού συστήματος.
  - Σε επίπεδο γλωσσών προγραμματισμού.
- → Κλασσικά προβλήματα ταυτοχρονισμού.

Το πρόβλημα του παραγωγού-καταναλωτή

- Το πρόβλημα του παραγωγού-καταναλωτή (producer-consumer) είναι ένα από τα κλασσικότερα προβλήματα ταυτοχρονισμού.
- Ένας ή περισσότεροι παραγωγοί παράγουν δεδομένα και τα αποθηκεύουν σε μία προσωρινή μνήμη (buffer).
- Ένας καταναλωτής αφαιρεί τα δεδομένα από τη μνήμη, ένα κάθε φορά.
- Μόνο ένας παραγωγός ή καταναλωτής μπορεί να έχει πρόσβαση στη μνήμη ανά πάσα στιγμή.
- Το πρόβλημα εδώ είναι να αποφευχθούν οι περιπτώσεις να προσπαθήσει ένας παραγωγός να αποθηκεύσει δεδομένα στη μνήμη όταν αυτή είναι γεμάτη ή να προσπαθήσει ο καταναλωτής να αφαιρέσει δεδομένα αν η μνήμη είναι άδεια.

97

Βασικός κώδικας

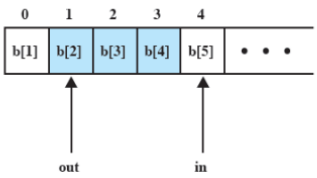
- Θεωρούμε ένα προσωρινό χώρο μνήμης b με άπειρες θέσεις μνήμης:

Παραγωγός	Καταναλωτής

`while (1)`
`{`
`/* produce item v */ ;`
`b[in] = v;`
`in++;`
`}`
	`while (1)`
	`{`
	`while (in <= out)`
	`/*do nothing */ ;`
	`w = b[out];`
	`out++;`
	`/* consume item w */ ;`
	`}`

98

Η δομή της κοινής προσωρινής μνήμης



Note: shaded area indicates portion of buffer that is occupied

Figure 5.11 Infinite Buffer for the Producer/Consumer Problem

99

Λάθος επίλυση του προβλήματος με δυαδικούς σημαφόρους

```
int n;
binary_semaphore s = 1,
                 delay = 0;

void main()
{
  n = 0;
  parbegin
     producer;
     consumer;
  parend
}
```

```
void producer()
{
 while (1) {
    produce();
    semWaitB(s);
    append();
    n++;
    if (n == 1) semSignalB(delay);
                                /* buffer was empty, so */
    semSignalB(s);              /* activate consumer */
 }
}

void consumer()
{
 semWaitB(delay); /* wait until buffer not empty */
 while (1) {
    semWaitB(s);
    take();
    n--;
    semSignalB(s);
    consume();
    if (n == 0) semWaitB(delay); */ buffer is empty */
 }
}
```


100

Επεξήγηση της λύσης

- Η ιδέα της λύσης είναι η χρήση της μεταβλητής `n` (=`in-out`) για την υλοποίηση των δύο δεικτών `in` και `out`.
- Επίσης, ο σημαφόρος `s` επιβάλει τον αμοιβαίο αποκλεισμό και ο σημαφόρος `delay` αναγκάζει τον καταναλωτή να περιμένει αν η μνήμη είναι άδεια.

101

Πιθανή εκτέλεση των εμπλεκόμενων διεργασιών

Table 5.4 Possible Scenario for the Program of Figure 5.9

	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	if (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	if (n==1) (semSignalB(delay))		0	1	1
13	semSignalB(s)		1	1	1
14		if (n==0) (semWaitB(delay))	1	1	1
15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1
18		if (n==0) (semWaitB(delay))	1	0	0
19		semWaitB(s)	0	0	0
20		n--	0	-1	0
21		semSignlaB(s)	1	-1	0

NOTE: White areas represent the critical section controlled by semaphore s.

102

Γιατί η λύση αυτή είναι λανθασμένη

- Όταν ο καταναλωτής έχει αφαιρέσει όλα τα δεδομένα από τη μνήμη, η εντολή `if` τον θέτει υπό αναστολή.
- Στη γραμμή 14 όμως δεν ικανοποιείται η συνθήκη της `if` παρόλο που ο καταναλωτής στη γραμμή 8 είχε μηδενίσει την τιμή της μεταβλητής `n`.
- Ο λόγος είναι ότι ο παραγωγός είχε αυξήσει την τιμή της `n` πριν ο καταναλωτής την εξετάσει στη γραμμή 14.
- Το αποτέλεσμα είναι η `n` να πάρει τιμή -1 που σημαίνει ότι ο καταναλωτής πήρε δεδομένα από άδεια μνήμη.
- Τα πρόβλημα δεν επιλύεται με τη μεταφορά της εντολή `if` μέσα στο κρίσιμο τμήμα του καταναλωτή (στην ομάδα των εντολών που περικλείονται από τις εντολές `semWaitB(s)` ... `semSignalB(s)`) γιατί τότε θα είχαμε αδιέξοδο (π.χ. μετά την γραμμή 8 δεν θα εκτελείτο ποτέ το `semSignalB(s)`).
- Για την επίλυση του προβλήματος χρειάζεται η εισαγωγή μίας τοπικής μεταβλητής στον καταναλωτή και να της δίνεται μέσα στο κρίσιμο τμήμα η τιμή της `n`.

Σωστή επίλυση του προβλήματος με δυαδικούς σημαφόρους

```
int n;
binary_semaphore s = 1, delay = 0;

void main()
{
  n = 0;
  parbegin
     producer;
     consumer;
  parend
}
```

```
void producer()
{
 while (1) {
    produce();
    semWaitB(s);
    append();
    n++;
    if (n == 1) semSignalB(delay);
    semSignalB(s);
 }
}

void consumer()
{
 int m; /* local variable */
 semWaitB(delay);
 while (1) {
    semWaitB(s);
    take();
    n--;
    m = n;
    semSignalB(s);
    consume();
    if (m == 0) semWaitB(delay);
 }
}
```


Επίλυση του προβλήματος με γενικούς σημαφόρους

```
semaphore n = 0, s = 1;

void main()
{
 parbegin
    producer;
    consumer;
 parend
}
```

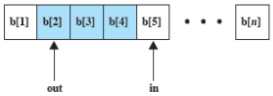
```
void producer()
{
 while (1) {
    produce();
    semWait(s);
    append();
    semSignal(s);
    semSignal(n);
 }
}

void consumer()
{
 while (1) {
    semWait(n);
    semWait(s);
    take();
    semSignal(s);
    consume();
 }
}
```

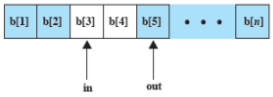

Επεξήγηση της λύσης

- Με χρήση γενικών σημαφόρων, η λύση είναι κατά τι πιο απλή.
- Η μεταβλητή `n` έχει αντικατασταθεί με ένα σημαφόρο αλλά η τιμή της παραμένει ίση με το πλήθος των δεδομένων στη μνήμη.
- Ακόμα και αν κατά λάθος οι εντολές `semSignal(s)` και `semSignal(n)` εκτελεσθούν αντίστροφα (που σημαίνει ότι η εντολή `semSignal(n)` θα εκτελεσθεί μέσα στο κρίσιμο τμήμα του παραγωγού χωρίς να διακοπεί είτε από κάποιον άλλο παραγωγό είτε από τον καταναλωτή), το πρόγραμμα θα εκτελεσθεί κανονικά διότι ο καταναλωτής πρέπει να εξετάσει και τους δύο σημαφόρους πριν εισέλθει στο κρίσιμο τμήμα του.

Παραλλαγή του προβλήματος με μνήμη πεπερασμένων θέσεων



(a)



(b)

Figure 5.15 Finite Circular Buffer for the Producer/Consumer Problem

Βασικός κώδικας με μνήμη πεπερασμένης χωρητικότητας

Παραγωγός	Καταναλωτής

`while (1)`
`{`
`/* produce item v */`
`while ((in + 1) % n == out)`
`/* do nothing */;`
`b[in] = v;`
`in = (in + 1) % n`
`}`
	`while (1)`
	`{`
	`while (in == out)`
	`/* do nothing */;`
	`w = b[out];`
	`out = (out + 1) % n;`
	`/* consume item w */`
	`}`

Επίλυση του προβλήματος με γενικούς σημαφόρους

```
const int sizeofbuffer =
                /* buffer size */;
semaphore s = 1, n = 0,
          e = sizeofbuffer;

void main()
{
 parbegin
    producer;
    consumer;
 parend
}
```

```
void producer()
{
 while (1) {
    produce();
    semWait(e);
    semWait(s);
    append();
    semSignal(s);
    semSignal(n);
 }
}

void consumer()
{
 while (1) {
    semWait(n);
    semWait(s);
    take();
    semSignal(s);
    semSignal(e);
    consume();
 }
}
```


109

Επίλυση του προβλήματος με παρακολουθητή — 1

```
monitor boundedbuffer()
{
 char buffer[N]; /* space for N items */
 int nextin, nextout; /* buffer pointers */
 int count; /* number of items in buffer */
 cond notfull, notempty; /* condition variables for synchronization */

 void append(char x)
 {
  if (count == N) cwait(notfull); /* buffer is full; avoid overflow */
  buffer[nextin] = x;
  nextin = (nextin + 1) % N;
  count++; /* one more item in buffer */
  csignal(notempty); /* resume any waiting consumer */
 }

 void take(char x)
 {
  if (count == 0) cwait(notempty); /* buffer is empty; avoid underflow */
  x = buffer[nextout];
  nextout = (nextout + 1) % N;
  count--; /* one fewer item in buffer */
  csignal(notfull); /* resume any waiting producer */
 }
 /* monitor body */
 nextin = 0; nextout = 0; count = 0; /* buffer initially empty */
}
```


110

Επίλυση του προβλήματος με παρακολουθητή — 2

```
void producer()
{
 char x;
 while (1) {
    produce(x);
    boundedbuffer.append(x);
 }
}

void consumer()
{
 char x;
 while (1) {
    boundedbuffer.take(x);
    consume(x);
 }
}

void main()
{
 parbegin producer; consumer; parend
}
```


111

Παραλλαγή του κώδικα με χρήση `cnotify`

```
void append(char x)
{
 while (count == N) cwait(notfull); /* buffer is full; avoid overflow */
 buffer[nextin] = x;
 nextin = (nextin + 1) % N;
 count++; /* one more item in buffer */
 cnotify(notempty); /* notify any waiting consumer */
}

void take(char x)
{
 while (count == 0) cwait(notempty); /* buffer is empty; avoid underflow */
 x = buffer[nextout];
 nextout = (nextout + 1) % N;
 count--; /* one fewer item in buffer */
 cnotify(notfull); /* notify any waiting producer */
}
```

- Σημειώστε τη χρήση της εντολής `while` αντί της `if`. Επειδή η `cnotify` δεν εκτελείται αμέσως και κατ' επέκταση δεν υπάρχει εγγύηση ότι κάποια άλλη διεργασία δεν θα προλάβει να εισέλθει στον παρακολουθητή πριν από αυτήν την διεργασία που περιμένει, η τελευταία αυτή χρειάζεται να επανελέγξει αν συνεχίζει να ισχύει η συνθήκη.

112

Επίλυση του προβλήματος με μηνύματα

```
const int capacity =
          /* buffering capacity */ ;
message null = /* empty message */ ;
int i;

void main()
{
 create_mailbox (mayproduce);
 create_mailbox (mayconsume);

 for (int i = 1; i <= capacity; i++)
     send(mayproduce,null);

 parbegin
    producer;
    consumer;
 parend
}
```

```
void producer()
{
 message pmsg;
 while (1) {
    receive(mayproduce,pmsg);
    pmsg = produce();
    send(mayconsume,pmsg);
 }
}

void consumer()
{
 message cmsg;
 while (1) {
    receive(mayconsume,cmsg);
    consume(cmsg);
    send(mayproduce,null);
 }
}
```


113

Το πρόβλημα των αναγνωστών-εγγραφέων

- Το πρόβλημα των αναγνωστών-εγγραφέων (readers-writers) μοντελοποιεί την πρόσβαση σε μία βάση δεδομένων.
- Υπάρχει μία ομάδα από διεργασίες που προσπαθούν να διαβάσουν ταυτόχρονα μία περιοχή μνήμης και μία ομάδα από διεργασίες που προσπαθούν να γράψουν ταυτόχρονα στην περιοχή αυτή.
- Μόνο μία διεργασία μπορεί ανά πάσα στιγμή να γράφει αλλά πολλές να διαβάζουν.
- Την ώρα που μία διεργασία γράφει καμία άλλη δεν μπορεί να διαβάζει.

114

Επίλυση του προβλήματος με σημαφόρους (οι αναγνώστες έχουν προτεραιότητα)

```
int readcount;
semaphore x = 1, wsem = 1;

void main()
{
 readcount = 0;
 parbegin
    reader;
    writer;
 parend
}
```

```
void reader()
{
 while (1) {
    semWait(x);
    readcount++;
    if (readcount == 1)
       semWait(wsem);
    semSignal(x);
    READUNIT();
    semWait(x);
    readcount--;
    if (readcount == 0)
       semSignal(wsem);
    semSignal(x);
 }
}

void writer()
{
 while (1) {
    semWait(wsem);
    WRITEUNIT();
    semSignal(wsem);
 }
}
```

115

115

Επεξήγηση και μειονεκτήματα της λύσης

- Γίνεται χρήση δύο σημαφόρων:
  - Ο `wsem` προστατεύει την κοινή περιοχή για γράψιμο ή διάβασμα.
  - Ο `x` προστατεύει την πρόσβαση στην κοινή μεταβλητή `readcount`.
- Η λύση αυτή δίνει προτεραιότητα στους αναγνώστες με την έννοια ότι αν έχει εισέλθει στην κοινή περιοχή ένας αναγνώστης, τότε όσο θα έρχονται αναγνώστες δεν θα είναι δυνατόν σε εγγραφείς να χρησιμοποιήσουν την περιοχή.
- Ο λόγος που αυτό είναι μειονέκτημα είναι γιατί συνήθως σε τέτοια σενάρια έχουμε λίγους εγγραφείς και πολλούς αναγνώστες.
- Επομένως, η λύση αυτή είναι δυνατόν να δημιουργήσει παρατεταμένη στέρηση στους εγγραφείς.

116

116

Επίλυση του προβλήματος με σημαφόρους (οι εγγραφείς έχουν προτεραιότητα)

```
int readcount, writecount;
semaphore x = 1, y = 1, z = 1,
          wsem = 1, rsem = 1;

void writer()
{
 while (1) {
    semWait(y);
    writecount++;
    if (writecount == 1)
       semWait(rsem);
    semSignal(y);
    semWait(wsem);
    WRITEUNIT();
    semSignal(wsem);
    semWait(y);
    writecount--;
    if (writecount == 0)
       semSignal(rsem);
    semSignal(y);
 }
}
```

```
void main()
{
 readcount = writecount = 0;
 parbegin reader; writer; parend
}

void reader()
{
 while (1) {
    semWait(z);
    semWait(rsem);
    semWait(x);
    readcount++;
    if (readcount == 1)
       semWait(wsem);
    semSignal(x);
    semSignal(rsem);
    semSignal(z);
    READUNIT();
    semWait(x);
    readcount--;
    if (readcount == 0)
       semSignal(wsem);
    semSignal(x);
 }
}
```

117

117

Συμπεριφορά των σημαφόρων

Readers only in the system	•*wsem* set •no queues
Writers only in the system	•*wsem* and *rsem* set •writers queue on *wsem*
Both readers and writers with read first	•*wsem* set by reader •*rsem* set by writer •all writers queue on *wsem* •one reader queues on *rsem* •other readers queue on *z*
Both readers and writers with write first	•*wsem* set by writer •*rsem* set by writer •writers queue on *wsem* •one reader queues on *rsem* •other readers queue on *z*

118

118

Επίλυση του προβλήματος με παρακολουθητή — 1

```
monitor readers_writers()
{
 int reader_count = 0, /* number of readers in critical section */
     writer_count = 0; /* number of writers waiting to enter critical section */
 bool busy = false;    /* checks if a writer is in the critical section */
 cond ok_to_read, ok_to_write;

 void start_read()
 {
  if (busy || writer_count > 0)
     wait(ok_to_read);
  reader_count++;
  signal(ok_to_read);
 }

 void start_write()
 {
  if (busy || reader_count > 0)
    {
      writer_count++;
      wait(ok_to_write);
    }
  busy = true;
 }
```

```
 void stop_read()
 {
  reader_count--;
  if (reader_count == 0) signal(ok_to_write);
 }

 void stop_write()
 {
  busy = false;
  if (writer_count > 0)
    {
     writer_count--;
     signal(ok_to_write);
    }
  else signal(ok_to_read);
 }
```

119

119

Επίλυση του προβλήματος με παρακολουθητή — 2

```
void reader()
{
 while (1) {
    readers_writers.start_read();
    READ();
    readers_writers.stop_read();
 }
}

void writer()
{
 while (1) {
    make_data(&info);            /* create new information to add */
    readers_writers.start_write();
    WRITE(info);
    readers_writers.stop_write();
 }
}

void main()
{
 parbegin reader; writer; parend
}
```

120

120

Επίλυση του προβλήματος με μηνύματα — 1

```
void reader(int i)
{
 message rmsg;
 while (1) {
    rmsg = i;
    send(readrequest,rmsg);
    receive(mbox[i],"OK");
    READUNIT();
    rmsg = i;
    send(finished,rmsg);
 }
}
```

```
void writer(int j)
{
 message rmsg;
 while (1) {
    rmsg = j;
    send(writerequest,rmsg);
    receive(mbox[j],"OK");
    WRITEUNIT();
    rmsg = j;
    send(finished,rmsg);
 }
}
```

121

121

Επίλυση του προβλήματος με μηνύματα — 2

```
void controller()
{
 while (1)
 {
  if (count > 0) {
     if (!empty(finished)) {
        receive(finished,"finished");
        count++;
     }
     else if (!empty(writerequest)) {
            receive(writerequest,msg);
            writer_id = msg.id;
            count = count – 100;
        }
     else if (!empty(readrequest)) {
            receive(readrequest,msg);
            count--;
            send(msg.id,"OK");
        }
 }
```

```
 if (count == 0) {
    send(writer_id,"OK");
    receive(finished,msg);
    count = 100;
 }
 while (count < 0) {
    receive(finished,msg);
    count++;
 }
}
```

122

122

Επεξήγηση της λύσης — 1

- Η κοινή μνήμη ελέγχεται από μία διεργασία (`controller`).
- Αν ένας εγγραφέας ή αναγνώστης θέλει να έχει πρόσβαση στη μνήμη, στέλνει μία αίτηση στον ελεγκτή, παίρνει πίσω άδεια με ένα μήνυμα `OK` και όταν ολοκληρώσει την πρόσβαση στέλνει στον ελεγκτή το μήνυμα `finished`.
- Ο ελεγκτής έχει τρία γραμματοκιβώτια, ένα για κάθε κατηγορία μηνυμάτων που δύναται να δεχθεί.
- Ο ελεγκτής δίνει προτεραιότητα στα μηνύματα που προέρχονται από εγγραφείς.

123

123

Επεξήγηση της λύσης — 2

- Η χρήση της μεταβλητής `count` με τον τρόπο που δείχνει ο κώδικας υλοποιεί τον αμοιβαίο αποκλεισμό.
- Συγκεκριμένα, στη `count` δίνεται ως αρχική τιμή ένας αριθμός μεγαλύτερος από τον μέγιστο αριθμό αναγνωστών (π.χ. 100), και ο `controller` λειτουργεί ως εξής:
  - Αν `count>0`, δεν υπάρχουν εγγραφείς σε αναμονή αλλά μπορεί να υπάρχουν αναγνώστες. Εξυπηρέτησε πρώτα τα μηνύματα `finished` και μετά αιτήσεις για γράψιμο και διάβασμα, με αυτή τη σειρά.
  - Αν `count=0`, υπάρχει μόνο ένας εγγραφέας και εξυπηρετείται.
  - Αν `count<0`, υπάρχει ένας εγγραφέας που έχει κάνει αίτηση για να εισέλθει στο κρίσιμο τμήμα αλλά περιμένει λόγω ύπαρξης αναγνωστών σε αυτό. Σε αυτήν την περίπτωση, δεν θα εισέλθει άλλος αναγνώστης στο κρίσιμο τμήμα και μόλις όλοι οι αναγνώστες σε αυτό έχουν εξέλθει, θα εξυπηρετηθεί ο εγγραφέας.

124

124

Το πρόβλημα του κουρείου

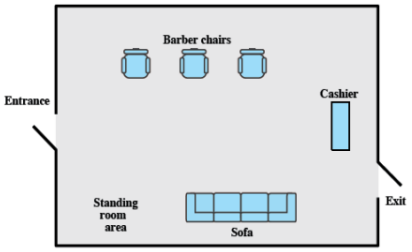


Figure A.1 The Barbershop

125

125

Βασικά στοιχεία του προβλήματος

- Μόνο 20 πελάτες μπορούν να βρίσκονται στο κουρείο ανά πάσα στιγμή.
- Μόνο 4 πελάτες μπορούν να κάθονται στον καναπέ.
- Μόνο 3 πελάτες μπορούν να κουρεύονται ταυτόχρονα.
- Ο πελάτης πρώτα μπαίνει στην αίθουσα και στέκεται όρθιος (αν ο καναπές είναι γεμάτος), μετά κάθεται στον καναπέ και κάποια στιγμή κουρεύεται από κάποιον από τους κουρείς.
- Μετά το κούρεμα, ο πελάτης πληρώνει στο μοναδικό ταμείο που υπάρχει (ο κουρέας που τον εξυπηρέτησε υιοθετεί προσωρινά το ρόλο του ταμία) πριν εξέλθει του κουρείου.
- Αν κάποιος από τους κουρείς δεν έχει προσωρινά δουλειά, τότε κοιμάται στην καρέκλα του.

126

126

Λύση με σημαφόρους — 1

```
semaphore max_capacity = 20,
          sofa = 4,
          barber_chair = 3,
          coord = 3,
          cust_ready = 0,
          finished = 0,
          leave_b_chair = 0,
          payment= 0,
          receipt = 0;

void main()
{
 parbegin
    customer;
    . . . 50 times, . . .
    customer;
    barber;
    barber;
    barber;
    cashier;
 parend
}
```


127

Λύση με σημαφόρους — 2

```
void customer ()
{
 semWait(max_capacity);
 enter_shop();
 semWait(sofa);
 sit_on_sofa();
 semWait(barber_chair);
 get_up_from_sofa();
 semSignal(sofa);
 sit_in_barber_chair();
 semSignal(cust_ready);
 semWait(finished);
 leave_barber_chair();
 semSignal(leave_b_chair);
 pay();
 semSignal(payment);
 semWait(receipt);
 exit_shop();
 semSignal(max_capacity)
}
```

```
void barber()
{
 while (1) {
    semWait(cust_ready);
    semWait(coord);
    cut_hair();
    semSignal(coord);
    semSignal(finished);
    semWait(leave_b_chair);
    semSignal(barber_chair);
 }
}
void cashier()
{
 while (1) {
    semWait(payment);
    semWait(coord);
    accept_pay();
    semSignal(coord);
    semSignal(receipt);
 }
```


128

Ρόλος των σημαφόρων

Semaphore	Wait Operation	Signal Operation
max_capacity	Customer waits for space to enter shop.	Exiting customer signals customer waiting to enter.
sofa	Customer waits for seat on sofa.	Customer leaving sofa signals customer waiting for sofa.
barber_chair	Customer waits for empty barber chair.	Barber signals when that barber's chair is empty.
cust_ready	Barber waits until a customer is in the chair.	Customer signals barber that customer is in the chair.
finished	Customer waits until his haircut is complete.	Barber signals when done cutting hair of this customer.
leave_b_chair	Barber waits until customer gets up from the chair.	Customer signals barber when customer gets up from chair.
payment	Cashier waits for a customer to pay.	Customer signals cashier that he has paid.
receipt	Customer waits for a receipt for payment.	Cashier signals that payment has been accepted.
coord	Wait for a barber resource to be free to perform either the hair cutting or cashiering function.	Signal that a barber resource is free.

129

Προβλήματα με τη λύση

- Ένα πρώτο πρόβλημα, που προέρχεται από το συντονισμό με τον σημαφόρο `finished`, είναι ότι οι 3 πελάτες που κουρεύονται πρέπει να ολοκληρώσουν το κούρεμα με τη σειρά που κάθισαν στις καρέκλες.
- Αυτό σημαίνει ότι αν ένας κουρέας είναι πιο γρήγορος από τους άλλους ή ένας πελάτης χρειάζεται λιγότερο κούρεμα, θα δημιουργηθεί πρόβλημα με κάποιον πελάτη είτε να σηκώνεται από την καρέκλα του πριν ολοκληρώσει το κούρεμα είτε να περιμένει καθισμένος σε αυτή αν και έχει τελειώσει το κούρεμα.
- Επίσης, αν περιμένουν για να πληρώσουν περισσότεροι από ένας πελάτες, ο κουρέας που παίζει το ρόλο του ταμία μπορεί να πληρωθεί από έναν πελάτη αλλά να αφήσει άλλον να φύγει.
- Επιπλέον υπάρχουν περιπτώσεις που ο αμοιβαίος αποκλεισμός στις 3 καρέκλες δεν λειτουργεί σωστά.
- Τέλος, ένας πελάτης είναι υποχρεωμένος πρώτα να καθίσει στον καναπέ και μετά σε κάποια καρέκλα για να κουρευτεί ακόμα και αν το κουρείο είναι άδειο.
- Δείτε το παράρτημα Α για λεπτομερή ανάλυση του προβλήματος.

130

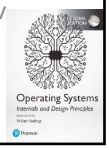
ΕΠΛ222: Λειτουργικά Συστήματα

(μετάφραση στα ελληνικά των διαφανειών του βιβλίου Operating Systems: Internals and Design Principles, 9/E, William Stallings)

Τέλος Ενότητας 4

Οι διαφάνειες αυτές έχουν συμπληρωματικό και επεξηγηματικό χαρακτήρα και σε καμία περίπτωση δεν υποκαθιστούν το βιβλίο

Γιώργος Α. Παπαδόπουλος
Τμήμα Πληροφορικής
Πανεπιστήμιο Κύπρου



131