ΜΥΥ601 Λειτουργικά Συστήματα

Εαρινό 2024

Μάθημα 4

Ταυτοχρονισμός: Αμοιβαίος Αποκλεισμός

Τμήμα Μηχανικών Η/Υ & Πληροφορικής
Πανεπιστήμιο Ιωαννίνων

1

Περίγραμμα

- Εισαγωγή
- Αμοιβαίος Αποκλεισμός
- Μηχανισμοί Λογισμικού
- Μηχανισμοί Υλικού

2

Εισαγωγή

- Ζητήματα ταυτοχρονισμού
  - Επικοινωνία μεταξύ διεργασιών
  - Κοινοχρησία και ανταγωνισμός για πόρους
  - Συγχρονισμός δραστηριοτήτων
  - Κατανομή χρόνου επεξεργαστή
- Εμφάνιση ταυτοχρονισμού
  - Πολυπρογραμματισμός – ένας επεξεργαστής
    - Πολλαπλές ενεργές εφαρμογές
    - Δομή εφαρμογής με πολλαπλές διεργασίες
    - Εσωτερική δομή λειτουργικού συστήματος με πολλαπλές διεργασίες
  - Πολυεπεξεργασία – πολλαπλοί επεξεργαστές
  - Κατανεμημένος υπολογισμός – πολλαπλές κατανεμημένες μηχανές

3

Προβλήματα

- Ταυτόχρονη επεξεργασία
  - Πολυπρογραμματισμός – εναλλαγή διεργασιών
  - Πολυεπεξεργασία – χρονική επικάλυψη διεργασιών
- Δυσκολίες που μπορούν να προκύψουν
  - Κοινοχρησία σφαιρικών πόρων
    - Σημαντική η σειρά ανάγνωσης/εγγραφής των μεταβλητών
  - Προβληματική εκχώρηση πόρων
    - Δέσμευση καναλιού Ε/Ε κατά τη διάρκεια αναστολής μιας διεργασίας
  - Δυσκολία εκσφαλμάτωσης κώδικα
    - Μη επαναλαμβανόμενη εκτέλεση
    - Μη προκαθορισμένα αποτελέσματα

4

Παράδειγμα - Πολυπρογραμματισμός

- Θεωρείστε
  - Πολυπρογραμματισμό σε έναν επεξεργαστή
  - Δύο διεργασίες P1 και P2
  - Κοινόχρηστη συνάρτηση echo()
- Ακολουθία
  - P1 καλεί echo()
    - chin = 'x';
  - P1 διακόπτεται
  - P2 καλεί echo()
    - chin = 'y'; chout = 'y'; εμφάνιση 'y'
  - P1 επανέρχεται
    - chout = chin; εμφάνιση 'y'
  - Το 'y' εμφανίζεται δύο φορές (P1 και P2)
  - Το 'x' δεν εμφανίζεται ποτέ !
- Λύση
  - Κλήση echo() από μία διεργασία κάθε φορά

```
void echo() {

    /* get char from keyboard */
        chin = getchar();

    /* copy char to chout */
        chout = chin;

        /* display char */
        putchar(chout);
}
```


5

Παράδειγμα - Πολυεπεξεργασία

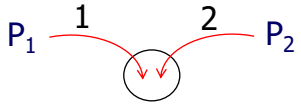
Διεργασία P1	Διεργασία P2
.	.
chin = getchar(); /* 'x' */	.
.	chin = getchar(); /* 'y' */
chout = chin;	chout = chin;
putchar(chout);	.
.	putchar(chout);

- Θεωρείστε
  - Πολυεπεξεργαστικό σύστημα
  - Δύο διεργασίες P1 και P2 τρέχουν σε διαφορετικούς επεξεργαστές παράλληλα
  - Κοινόχρηστη συνάρτηση echo()
- Ακολουθία εντολών
  - Το 'y' εμφανίζεται δύο φορές (P1 και P2)
  - Το 'x' δεν εμφανίζεται ποτέ !
- Λύση: επιτρέψτε μόνο μία διεργασία να καλέσει echo() κάθε φορά

6

Συνθήκη Ανταγωνισμού (Race Condition)

- Ορισμός
  - Πολλαπλές διεργασίες ή νήματα διαβάζουν και ενημερώνουν την ίδια συλλογή δεδομένων
  - Οι τελικές τιμές των δεδομένων εξαρτώνται από τη σειρά εκτέλεσης των εντολών των διεργασιών ή νημάτων
- Παράδειγμα
  - Έστω διεργασίες P_1 και P_2 που ενημερώνουν μια κοινόχρηστη μεταβλητή
  - Αν η P_1 γράψει την τιμή 1 και η P_2 την τιμή 2 τότε η τελική τιμή της μεταβλητής εξαρτάται από το ποια διεργασία έγραψε τελευταία



7

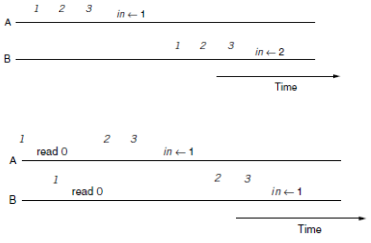
Δεύτερη Συνθήκη Ανταγωνισμού

- Έστω εντολή που προσαυξάνει έναν μετρητή δείκτη κατά ένα

$in \leftarrow in + 1$

```
1  LOAD in, R0      // Load the value of in into a register
2  ADD R0, 1        // Increment
3  STORE R0, in     // Store result back to in
```

- Σωστή εκτέλεση
  - Εκτέλεση των εντολών της Α πριν τις εντολές διεργασίας Β
- Λάθος εκτέλεση
  - Ανάμικτη ακολουθία εκτέλεσης
  - Τελική τιμή *in* 1 αντί της 2



8

Πρόβλημα: Αμοιβαίος Αποκλεισμός

- Κρίσιμος πόρος
  - Μοναδικός πόρος που δε μπορεί να χρησιμοποιηθεί ταυτόχρονα από πολλές διεργασίες
- Κρίσιμη περιοχή
  - Μέρος προγράμματος που χρησιμοποιεί έναν κρίσιμο πόρο
- Αμοιβαίος αποκλεισμός
  - Επιτρέπει μία διεργασία κάθε φορά να τρέξει στην κρίσιμη περιοχή
- Παράδειγμα
  - Θεωρήστε δύο διεργασίες που προσπαθούν να κάνουν χρήση εκτυπωτή
  - Αν επιτραπούν να τρέξουν ταυτόχρονα, παίρνουμε μπλεγμένη εκτύπωση
  - Λύση: επιτρέπουμε μόνο μία διεργασία κάθε φορά να εκτυπώσει

Αφαίρεση Προβλήματος

- Σκοπός
  - Έκφραση του προβλήματος
  - Χρήση αφηρημένων όρων
  - Αξιοποίηση υποστήριξης ΛΣ
- parbegin(P(1), ..., P(n))
  - Αναστέλλει την κύρια διεργασία
  - Ξεκινάει τις P(1), ..., P(n)
  - Πόρος: R
  - Περιμένει τις P να τερματίσουν
  - Επανέρχεται στην κύρια διεργασία
- enter/exitcritical(R)
  - Εξασφαλίζει αμοιβαίο αποκλεισμό
  - Η ταυτότητα πόρου παράμετρος R
  - Υποστηρίζεται από το ΛΣ

```
/* program mutual exclusion */
const int n = /* # processes */;
void P(int i) {
    while (true) {
        entercritical(R);
        /* critical section */
        exitcritical(R);
        /* other code */
    }
}

void main() {
    parbegin( P(1), ... P(n) );
}
```

Προβλήματα: Αδιέξοδο και Στέρηση

- Αδιέξοδο
  - Δυο διεργασίες P1 και P2 χρειάζονται τους πόρους R1 και R2
  - Κάθε διεργασία περιμένει για τον πόρο που έχει η άλλη διεργασία
  - Π.χ. η P1 έχει τον R1 και περιμένει τον R2,
    η P2 έχει τον R2 και περιμένει τον R1

- Στέρηση
  - Θεωρήστε τρεις διεργασίες P1, P2, P3
  - Όλες οι διεργασίες χρησιμοποιούν περιοδικά τον πόρο R
  - Το ΛΣ δίνει τον R στην P3 μετά την P1 και στην P1 μετά την P3
  - Η P2 δεν παίρνει ποτέ τον πόρο R παρόλο που δεν υπάρχει αδιέξοδο

Πρόβλημα: Συνέπεια Δεδομένων

- Έστω αντικείμενα που ικανοποιούν κάποια συνθήκη
  - Π.χ. a = b
- Αν η επεξεργασία σειριακά, τα δεδομένα συνεπή
  - Π.χ. από a = b =1 παίρνουμε a = b = 4
- Ταυτόχρονη επεξεργασία αφήνει ασυνεπή δεδομένα
  - Αμοιβαίος αποκλεισμός χωριστά στα a, b δεν αρκεί
  - Π.χ. τα a = b = 1 μπορούν να γίνουν a = 4, b =3
- Λύση
  - Δηλώνουμε κρίσιμη περιοχή τον κώδικα κάθε διεργασίας

P1: a = a + 1;
b = b + 1;

P2: b = 2 * b;
a = 2 * a;

Ταυτόχρονα:
a = a + 1;
b = 2 * b;
b = b + 1;
a = 2 * a;

Αλληλεπίδραση Διεργασιών

- Ανταγωνισμός
  - Οι διεργασίες αγνοούν η μία την άλλη
  - Π.χ. πολυπρογραμματισμός
  - Προβλήματα: αμοιβαίος αποκλεισμός, αδιέξοδο, στέρηση
- Συνεργασία με κοινοχρησία
  - Οι διεργασίες έμμεσα γνωρίζουν η μία την άλλη
  - Π.χ. διεργασίες με κοινοχρησία δεδομένων σε ενδιάμεση μνήμη
  - Προβλήματα: αμοιβαίος αποκλεισμός, αδιέξοδο, στέρηση, συνέπεια
- Συνεργασία με επικοινωνία
  - Οι διεργασίες άμεσα γνωρίζουν η μία την άλλη
  - Π.χ. οι διεργασίες επικοινωνούν για να ολοκληρώσουν μια ενέργεια
  - Προβλήματα: αδιέξοδο, στέρηση

Απαιτήσεις Αμοιβαίου Αποκλεισμού

1. **Μία διεργασία κάθε φορά στην κρίσιμη περιοχή (ΚΠ)**
2. Μια διεργασία που σταματάει έξω από την ΚΠ
   - Δεν πρέπει να αλληλεπιδρά με άλλες διεργασίες
3. Αποφυγή αδιεξόδου ή στέρησης
   - Είσοδος διεργασίας στην ΚΠ σε πεπερασμένο χρόνο
4. Όταν δεν υπάρχει διεργασία στην κρίσιμη περιοχή
   - Θα πρέπει να επιτραπεί άμεση είσοδος σε μια διεργασία
5. Καμία υπόθεση για τη σχετική ταχύτητα διεργασιών
6. Μια διεργασία παραμένει στην ΚΠ πεπερασμένο χρόνο

- Λύσεις
  - Λογισμικό εφαρμογών
  - Ειδικές εντολές στο υλικό του επεξεργαστή
  - Από το λειτουργικό σύστημα ή τη γλώσσα προγραμματισμού

Αλγόριθμός Dekker: Πρώτη Προσπάθεια

- Βασική υπόθεση
  - Μία πρόσβαση σε θέση μνήμης κάθε φορά
  - Κάθε πρόσβαση στη μνήμη ολοκληρώνεται χωρίς πρόβλημα
- Ενεργός αναμονή
  - Μια διεργασία περιμένει μέχρι *turn = process #*
  - Ενώ περιμένει, σπαταλά χρόνο επεξεργαστή
- Η λύση μοιάζει με *συρρουτίνες (coroutines)*
  - Συνεργαζόμενα τμήματα προγράμματος
  - Εναλλάσσουν την εκτέλεση (με εντολές yield)
  - Περιορίζονται μέσα σε μία διεργασία
  - Ανεπαρκή για ταυτόχρονο υπολογισμό

```
int turn = 0; /* global */

/* Process 0 */
while (turn != 0)
    /* do nothing */;
/* critical section */
turn = 1;

/* Process 1 */
while (turn != 1)
    /* do nothing */;
/* critical section */
turn = 0;
```


Κριτική της Πρώτης Προσπάθειας

- Μειονεκτήματα
  - Οι διεργασίες εναλλάσσονται αυστηρά στην κρίσιμη περιοχή (ΚΠ)
  - Αν μια διεργασία αποτύχει, η άλλη μένει εκτός ΚΠ για πάντα, ανεξάρτητα αν η πρώτη διεργασία απέτυχε μέσα ή έξω από την ΚΠ
- Μεταβλητή *turn*
  - Αποθηκεύει μόνο τον κωδικό της διεργασίας που μπαίνει στην ΚΠ
  - Αν μια διεργασία αποτύχει, δε μπορεί να ενημερώσει την *turn* ξανά
- Πρέπει να γνωρίζουμε την κατάσταση και των δύο διεργασιών
  - Κάθε διεργασία πρέπει να έχει το δικό της «κλειδί» εισόδου
  - Όταν μια διεργασία αποτύχει, η άλλη θα πρέπει να μπορεί να μπει

Αλγόριθμος Dekker: Δεύτερη Προσπάθεια

- Εισάγουμε το διάνυσμα δυαδικών *flag*
  - Η P0 έχει το *flag[0]* και η P1 έχει το *flag [1]*
  - Η P0 γράφει στο *flag[0]* και διαβάζει το *flag[1]*
  - Η P1 γράφει στο *flag[1]* και διαβάζει το *flag[0]*
- Για να εισέλθει στην κρίσιμη περιοχή
  1. Μια διεργασία περιμένει το άλλο *flag = false* (⇒ η άλλη διεργασία εκτός κρίσιμης περιοχής)
  2. Η διεργασία θέτει το δικό της *flag = true*
  3. Η διεργασία εισέρχεται στην κρίσιμη περιοχή
  4. Όταν φεύγει, θέτει το δικό της *flag = false*

```
boolean flag[2] = /* global */
    {false, false};

/* Process 0 */
while (flag[1])
    /* do nothing */;
flag[0] = true;
/* critical section */
flag[0] = false;

/* Process 1 */
while (flag[0])
    /* do nothing */;
flag[1] = true;
/* critical section */
flag[1] = false;
```


Κριτική της Δεύτερης Προσπάθειας

- Βελτίωση από την πρώτη προσπάθεια
  - Αν μια διεργασία αποτύχει εκτός της κρίσιμης περιοχής (ΚΠ), η άλλη διεργασία μπορεί να εισέλθει όσο συχνά θέλει
- Μειονέκτημα
  - Αν μια διεργασία αποτύχει εντός της κρίσιμης περιοχής ή αφού θέσει το *flag = true,* η άλλη διεργασία δε θα ξαναμπεί
- Δεν εξασφαλίζει αμοιβαίο αποκλεισμό
  - Η P0 βρίσκει το *flag[1] = false* [διακόπτεται]
  - Η P1 βρίσκει το *flag[0] = false* [διακόπτεται]
  - Η P0 θέτει το *flag[0] = true* και μπαίνει στην ΚΠ [διακόπτεται]
  - Η P1 θέτει το *flag[1] = true* και μπαίνει στην ΚΠ [διακόπτεται]
  - Τώρα οι P0 και P1 μαζί στην κρίσιμη περιοχή – Λάθος !

Αλγόριθμος Dekker: Τρίτη Προσπάθεια

- Η δεύτερη προσπάθεια απέτυχε
  - Επειδή μια διεργασία άλλαξε το δικό της flag
  - Αφού η άλλη διεργασία το έλεγξε ΑΛΛΑ
  - Πριν η άλλη διεργασία εισέλθει στην ΚΠ
- Πιθανή λύση
  - Μετακίνηση του *flag = true* πριν το while-loop
- Μειονέκτημα
  - Αν μια διεργασία αποτύχει εντός της κρίσιμης περιοχής ή αφού θέσει το δικό της *flag = true,* η άλλη διεργασία δε θα ξαναμπεί στην ΚΠ

```
boolean flag[2] = /* global */
    {false, false};

/* Process 0 */
flag[0] = true;
while (flag[1])
    /* do nothing */;
/* critical section */
flag[0] = false;

/* Process 1 */
flag[1] = true;
while (flag[0])
    /* do nothing */;
/* critical section */
flag[1] = false;
```


Κριτική της Τρίτης Προσπάθειας

- Ο αμοιβαίος αποκλεισμός δουλεύει
  - Θεωρείστε ότι η P0 θέτει *flag[0] = true* (παρομοίως για την P1)
  - Αν η P1 εντός της κρίσιμης περιοχής
    - Η P0 θα αποκλειστεί στο while-loop
  - Αν η P1 εκτός της κρίσιμης περιοχής
    - Η P1 δε θα μπει στην κρίσιμη περιοχή μέχρι η P0 να μπει και να βγει
- Αδιέξοδο
  - Η P0 θέτει το *flag[0] = true* [διακόπτεται]
  - Η P1 θέτει το *flag[0] = true* [διακόπτεται]
  - Η P0 βρίσκει το *flag[1] = true* και περιμένει [διακόπτεται]
  - Η P1 βρίσκει το *flag[0] = true* και περιμένει [διακόπτεται]
  - Τώρα οι P0 και P1 περιμένουν για πάντα – Λάθος !

Αλγόριθμος Dekker: Τέταρτη Προσπάθεια

- Η τρίτη προσπάθεια απέτυχε με αδιέξοδο
  - Επειδή κάθε διεργασία θέτει το δικό της *flag*
  - Χωρίς να ξέρει το *flag* της άλλης
- Πιθανή λύση
  - Κάθε διεργασία μπορεί να μηδενίσει το *flag* της
  - Έτσι μια διεργασία υποχωρεί στην άλλη διεργασία
- Ενεργό αδιέξοδο (livelock)
  - Η P0 θέτει *flag[0] = true* [διακόπτεται]
  - Η P1 θέτει *flag[1] = true* [διακόπτεται]
  - Η P0 ελέγχει το *flag[1]* [διακόπτεται]
  - Η P1 ελέγχει το *flag[0]* [διακόπτεται]
  - Η P0 θέτει *flag[0] = false* [διακόπτεται]
  - Η P1 θέτει *flag[1] = false* [διακόπτεται]
  - ...
  - Οι P0 & P1 δε μπαίνουν ποτέ στην ΚΠ – Λάθος!

```
/* global */
boolean flag[2] = {false, false};

/* Process 0 */
flag[0] = true;
while (flag[1]) {
      flag[0] = false;
      /* delay */;
      flag[0] = true;
}
/* critical section */
flag[0] = false;

/* Process 1 */
flag[1] = true;
while (flag[0]) {
      flag[1] = false;
      /* delay */;
      flag[1] = true;
}
/* critical section */
flag[1] = false;
```


Ορολογία

- Αδιέξοδο (Deadlock)
  - Σύνολο διεργασιών περιμένουν για συνθήκη που δεν ισχύει ποτέ
  - Οι διεργασίες μένουν εκτός της κρίσιμης περιοχής για πάντα
- Ενεργό αδιέξοδο (Livelock)
  - Ενδεχόμενη ακολουθία εκτέλεσης για σύνολο διεργασιών
  - Καμιά διεργασία δεν εισέρχεται στην κρίσιμη περιοχή
  - Εναλλακτικές ακολουθίες εκτέλεσης λύνουν το πρόβλημα
- Στέρηση (Starvation)
  - Διεργασία περιμένει για ένα πόρο που συνεχώς γίνεται διαθέσιμος
  - ΑΛΛΑ ο πόρος δε δίνεται ποτέ στη θεωρούμενη διεργασία
  - Η διεργασία δεν είναι ποτέ σίγουρη ότι θα αποκτήσει τον πόρο (δηλαδή μπορεί και να τον αποκτήσει...)

Σωστός Αλγόριθμος του Dekker

```
boolean flag[2] = {false, false};
int turn = 1;
void P0() {
    while (true) {
       flag[0] = true; /* intention */
       while (flag[1])
          if (turn == 1) { /* defer */
               flag[0] = false
               while (turn==1)
                     /* do nothing */;
               flag[0] = true;
          }
       /* critical section */
       turn = 1; /* break livelock  */
       flag [0] = false; /* exit section */
   }
}
```

```
void P1() {
    while (true) {
       flag[1] = true; /* intention */
       while (flag[0])
          if (turn == 0) { /* defer */
               flag[1] = false
               while (turn==0)
                     /* do nothing */;
               flag[1] = true;
          }
       /* critical section */
       turn = 0; /* break livelock */
       flag [1] = false; /* exit section */
   }
}
void main() {
   parbegin(P0, P1); }
```


Αλγόριθμος του Peterson

```
boolean flag[2] = {false, false};
int turn = 1;
void P0() {
    while (true) {
       flag[0] = true; /* intention */
       turn = 1; /* break livelock */
        while (flag[1] && turn == 1)
             /* do nothing */;
        /* critical section */
        flag [0] = false; /* exit */
    }
}
```

```
void P1() {
    while (true) {
       flag[1] = true; /* intention */
       turn = 0; /* break livelock */
        while (flag[0] && turn == 0)
             /* do nothing */;
        /* critical section */
        flag [1] = false; /* exit */
    }
}
void main() {
   parbegin(P0, P1);
}
```

Ανάλυση του Αλγορίθμου του Peterson

- Εγγύηση αμοιβαίου αποκλεισμού
  - Έστω η P0 θέτει *flag[0] = true*
  - Αν η P1 εντός της ΚΠ
    - Το *flag[1]=true* αποκλείει την P0
  - Αν η P1 εκτός της ΚΠ
    - Το *flag[0]=true* αποκλείει την P1
- Άλλες ιδιότητες
  - Εύκολη γενίκευση για n > 2 διεργασίες
- Αποφυγή ενεργού αδιεξόδου
  - Η P0 περιμένει στο while-loop
    - *flag[1] = true* & *turn =1*
  1. Η P1 δε θέλει να μπει στην ΚΠ
     - Απαιτείται *flag[1] = false*
     - Αδύνατο γιατί *flag[1] = true*
  2. Η P1 περιμένει να μπει
     - Αδύνατο επειδή *turn = 1*
     - Η P1 θα έμπαινε χωρίς αναμονή
  3. Η P1 μονοπωλεί την ΚΠ
     - Η P1 θέτει *turn = 0* πριν μπει
     - Έτσι δίνει ευκαιρία στην P0

Απενεργοποίηση Διακοπών

- Μηχανή ενός επεξεργαστή
  - Εφικτή μόνο η εναλλαγή διεργασιών
  - Εμποδίζουμε τη διακοπή διεργασίας
  - Απενεργοποιούμε διακοπές υλικού
  - Χρησιμοποιούμε υπηρεσίες του πυρήνα
  - Εξασφαλίζουμε αμοιβαίο αποκλεισμό
- Κόστος
  - Εμποδίζει την εναλλαγή διεργασιών
  - Μειώνει την απόδοση εκτέλεσης
  - Δεν δουλεύει σε πολυεπεξεργαστές

```
while (true) {
    /* disable interrupts */
    /* critical section */
    /* enable interrupts */
    /* remainder */
}
```

Ειδικές Εντολής Μηχανής

- Επεξεργαστές πολυεπεξεργαστικής μηχανής
  - Έχουν πρόσβαση σε κοινόχρηστη κύρια μνήμη
  - Λειτουργούν ανεξάρτητα ως ομότιμες οντότητες
  - Η απενεργοποίηση διακοπών δεν εξασφαλίζει αμοιβαίο αποκλεισμό επειδή εκτελούνται παράλληλα πολλαπλές διεργασίες
- Σε επίπεδο υλικού
  - Μια θέση μνήμης αποκλείει πολλές προσβάσεις την ίδια στιγμή
  - Προσθέτουμε εντολή που υποστηρίζει δύο ατομικές ενέργειες
    - Αυτές οι ενέργειες ολοκληρώνονται σε έναν κύκλο εντολής
    - Δε μπορούν να διακοπούν από άλλη εντολή
      Π.χ. ανάγνωση & εγγραφή μιας θέσης μνήμης

Test and Set

- Ατομική εκτέλεση
  - True στην πρώτη που βρίσκει το i = 0 αφού θέσει i = 1
  - False μέχρι να γίνει το i = 0

```
boolean testset(int *i) {
    /* complete atomically */
    if (*i == 0) { /* first to enter */
          *i = 1; /* notify others */
          return true;
    } else  { /* other already in */
          return false;
    }
}
```

```
/* program mutual exclusion */
const int n = /* # processes */;
int bolt = 0;
void P(int i) {
    while (true) {
          while (testset(&bolt) == false)
                /* do nothing */;
          /* critical section */
          bolt = 0;
          /* remainder */
    }
}

void main() { parbegin(P(1),..., P(n));}
```

Swap

- Ατομική ανταλλαγή
  - Την παράμετρο register
  - ΜΕ την παράμετρο memory

```
void swap(     int *register,
               int *memory) {
   int temp;
   /* complete atomically */
   temp = *memory;
   *memory = *register;
   *register = temp;
}
```

```
/* program mutual exclusion */
const int n = /* # processes */;
int bolt = 0;
void P(int i) {
    int keyi;          /* bolt + ∑ keyi = n */
    while (true) {
          keyi = 1;
          while (keyi != 0)
                swap(&keyi, &bolt);
          /* critical section */
           swap(&keyi, &bolt);
          /* remainder */
     }
}
void main() { parbegin(P(1),..., P(n)); }
```


Ιδιότητες Ειδικών Εντολών Μηχανής

- Απαιτήσεις
  - Πρόσβαση πολλαπλών επεξεργαστών σε κοινόχρηστη μνήμη
- Προτερήματα
  - Εφαρμόζονται σε κάθε αριθμό από επεξεργαστές
  - Απλές και εύκολες στην επαλήθευση ότι δουλεύουν σωστά
  - Υποστηρίζουν πολλαπλές κρίσιμες περιοχές (1 μεταβλητή/ΚΠ)
- Μειονεκτήματα
  - Η ενεργή αναμονή καταναλώνει χρόνο επεξεργαστή
  - Στέρηση δυνατή όταν επιλέγεται διεργασία σε αναμονή
  - Αδιέξοδο όταν μια διεργασία υψηλής προτεραιότητας περιμένει διεργασία χαμηλής προτεραιότητας να μηδενίσει τη μεταβλητή, επειδή η δεύτερη δεν εκτελείται ποτέ

Αμοιβαίος Αποκλεισμός σε Pthreads

- Τα νήματα POSIX χρησιμοποιούν μεταβλητές mutex για να επιτύχουν αμοιβαίο αποκλεισμό

```
pthread_mutex_t mutex;                  /* declaration */
pthread_mutex_init(&mutex, NULL);       /* initialization */

pthread_mutex_lock(&mutex);             /* use */
  /* critical section */
pthread_mutex_unlock(&mutex);

pthread_mutex_destroy(&mutex);          /* deallocation */
```