Διεργασίες (Processes)

Διεργασία (process) ή καθήκον (task)

- στοιχειώδης οντότητα/δραστηριότητα υπολογισμού (processing entity/activity)
- *εκτέλεση ενός προγράμματος*
- ένα (κύριο) νήμα (thread)/ρεύμα ελέγχου/εκτέλεσης
- διεργασία ≠ πρόγραμμα
    - μία διεργασία μπορεί να περιλαμβάνει την εκτέλεση περισσοτέρων από ένα προγραμμάτων
    - ένα πρόγραμμα που μπορεί να περιλαμβάνεται σε περισσότερες από μία διεργασίες ονομάζεται *επανεισαγόμενο πρόγραμμα (reentrant program)* ή *αμιγής διαδικασία (pure procedure)*
      (π.χ. αφαίρεση αντικειμένων από λίστα,
      υπολογισμός τετραγωνικής ρίζας κ.ά.)
    - διεργασία = ακολουθία ενεργειών = δυναμική
    - πρόγραμμα = ακολουθία εντολών = στατικό

Διεργασίες

Ακολουθιακές διεργασίες (sequential processes)

- Ακολουθιακή εκτέλεση εντολών

Ταυτόχρονες διεργασίες (concurrent processes)

- Ένα σύνολο από ακολουθιακές διεργασίες που αλληλεπιδρούν μεταξύ τους
- είτε "(ψευδο)ταυτόχρονη" εκτέλεση σ' έναν επεξεργαστή
- είτε ταυτόχρονη εκτέλεση σε πολλούς επεξεργαστές
- ανήκουν στον ίδιο ή διαφορετικούς χρήστες
- συνεργαζόμενες (cooperating) για έναν κοινό στόχο
- ανταγωνιζόμενες (competing) για τους πόρους του συστήματος

Σχεδιασμός λειτουργικού συστήματος

- ταυτόχρονες διεργασίες = (κατα)μερισμός πόρων
- **Προβλήματα**
  - αμοιβαίος αποκλεισμός (mutual exclusion)
  - συγχρονισμός (synchronization)
  - αδιέξοδα (deadlocks)
  - διαδεργασιακή επικοινωνία (interprocess communication), δηλαδή επικοινωνία μεταξύ διεργασιών

Αμοιβαίος αποκλεισμός
(Mutual exclusion)

Μη (κατα)μεριζόμενοι (χρονικά) πόροι

Συνθήκες αγώνων δρόμου (race conditions)

- Αν περισσότερες από μια διεργασίες θέλουν να χρησιμοποιήσουν ένα μη (κατά)μεριζόμενο πόρο στο ίδιο χρονικό διάστημα, τότε
    - ΜΙΑ ΜΟΝΟ διεργασία πρέπει να μπορεί να τον χρησιμοποιήσει
    - ΟΙ ΑΛΛΕΣ πρέπει να ΠΕΡΙΜΕΝΟΥΝ, μ' άλλα λόγια:

Δεν πρέπει να συμπίπτει χρονικά η εκτέλεση των κρίσιμων περιοχών των διεργασιών

- κρίσιμο τμήμα (critical section)
  αυτό που προσπελάζει κάποιον μη (κατα)μεριζόμενο πόρο
- μη κρίσιμο τμήμα (non-critical section)
  αυτό που δεν προσπελάζει μη (κατα)μεριζόμενους πόρους

Αμοιβαίος αποκλεισμός

ΠΡΟΒΛΗΜΑ

- N διεργασίες
- Δεν πρέπει να συμπίπτει χρονικά η εκτέλεση κρίσιμων περιοχών δύο διεργασιών
- Καμία υπόθεση για τον αριθμό και την ταχύτητα των επεξεργαστών
- Μία διεργασία περιμένει ΜΟΝΟ στο μη κρίσιμο τμήμα της
- Μία διεργασία δεν μπορεί να προκαλέσει την αναμονή άλλων διεργασιών έξω από την κρίσιμη περιοχή της

ΙΔΙΟΤΗΤΕΣ ΜΙΑΣ ΚΑΛΗΣ ΛΥΣΗΣ

- **Όχι αδιέξοδα (deadlocks):** κάποια από τις διεργασίες πρέπει να κερδίσει τον πόρο
- **Όχι υποσιτισμός (starvation):** όλες οι διεργασίες πρέπει τελικά να αποκτήσουν τον πόρο
- **Μικρή αναμονή (wait):** σε περίπτωση έλλειψης ανταγωνισμού ο πόρος πρέπει να αποκτάται σε σύντομο χρονικό διάστημα.

Αμοιβαίος αποκλεισμός

ΙΔΕΑ 1

Κλείδωμα μιας μεταβλητής (one locked variable)

- ελεύθερος: μοναδική (κατα)μεριζόμενη μεταβλητή τύπου boolean που παριστάνει αν ο πόρος είναι ελεύθερος/απασχολημένος

```
process P1;
{
   . . .
   while (true)
   {
   . . .
   while (!ελεύθερος) ;
→  ελεύθερος := false;
   <κρίσιμη περιοχή1>;
   ελεύθερος := true;
   . . .
   }
} // P1
```

```
process P2;
{
   . . .
   while (true)
   {
   . . .
→  while (!ελεύθερος) ;
   ελεύθερος := false;
←<κρίσιμη περιοχή2>;
   ελεύθερος:= true;
   . . .
   }
} // P2
```

- αρχικά ελεύθερος =true
- Η διεργασία εξετάζει τη μεταβλητή ελεύθερος
- αν ελεύθερος ==true εισέρχεται στην ΚΠ της, αλλιώς περιμένει

Αμοιβαίος αποκλεισμός

ΙΔΕΑ 2

Κλείδωμα δύο μεταβλητών (two locked variables)

- σειράP1, σειράP2: δύο (κατα)μεριζόμενες μεταβλητές τύπου `boolean` που παριστάνουν αν είναι η σειρά της `P1` ή η σειρά της `P2` να πάρει τον πόρο

```
process P1;
{
   . . .
   while (true)
   {
    . . .
    while (σειράP2);
    σειράP1 = true;
    <κρίσιμη περιοχή>;
    σειράP1 = false;
    . . .
   }
} // P1
```

```
process P2;
{
   . . .
   while (true)
   {
    . . .
    while (σειράP1) ;
    σειράP2= true;
    <κρίσιμη περιοχή>;
    σειράP2 = false;
    . . .
   }
} // P2
```

- αρχικά `σειράP1 = σειράP2 =false`
- η `P1` (`P2`) εξετάζει την μεταβλητή σειράP2 (σειράP1)
  - `P1`: Αν `σειράP2 == true` τότε περιμένει, αλλιώς εισέρχεται στην ΚΠ της
  - `P2`: Αν `σειράP1 == true` τότε περιμένει, αλλιώς εισέρχεται στην ΚΠ της
- το ίδιο πρόβλημα και πάλι

Αμοιβαίος αποκλεισμός

ΙΔΕΑ 3

Αυστηρή εναλλαγή (Strict alternation)

- σειρά2: μία μοναδική (κατα)μεριζόμενη μεταβλητή τύπου boolean που παριστάνει ποιας διεργασίας είναι σειρά να προσπελάσει τον πόρο

```
process P1;
{
   . . .
   while (true)
   {
   . . .
   while (σειρά2) ;
   <κρίσιμη περιοχή>;
    σειρά2 := true;
   . . .
   }
} // P1
```

```
process P2;
{
   . . .
   while (true)
   {
   . . .
   while (!σειρά2) ;
   <κρίσιμη περιοχή>;
    σειρά2 = false;
   . . .
   }
} // P2
```

- αρχικά σειρά2=true // έστω
- εξασφαλίζει τον αμοιβαίο αποκλεισμό!

ΠΡΟΒΛΗΜΑΤΑ

- Η χρήση του πόρου εναλλάσσεται μεταξύ των P2 και P1 (P2, P1, P2, …)
- Κακή ιδέα όταν η μια από τις διεργασίες είναι πολύ αργή
  - η άλλη αποκτά τον πόρο με τη συχνότητα της βραδύτερης
  - Απασχολούμενη σε αναμονή (busy waiting)
    - η ΚΜΕ περιμένει εξετάζοντας συνεχώς τη τιμή της σειρά2

- Η λύση αυτή δε λειτουργεί χωρίς ανταγωνισμό.

Αμοιβαίος αποκλεισμός

- Ιδέες 1, 2
  - Πρόβλημα:

    εκχώρηση τιμής μεταβλητής ≠ εξέτασης τιμής

- Νέοι αλγόριθμοι λύνουν αυτό το πρόβλημα
  4. **Αλγόριθμος Dekker**
     - Πρώτη λύση λογισμικού
     - Πολύπλοκη λύση
     - Βλ. Άσκη. και Απάντ. 1.1.3 βιβλίου
  5. **Αλγόριθμος Peterson (1981)**
     - Η πρώτη ικανοποιητική λύση στο πρόβλημα
     - Βλέπε Άσκ. και Απάντ. 1.1.5 βιβλίου
  - **ΑΛΛΑ** και αυτοί είναι αλγόριθμοι απασχολούμενης αναμονής: ο χρόνος της ΚΜΕ σπαταλιέται σε συνεχή έλεγχο των τιμών μιας ή περισσοτέρων μεταβλητών

Σηματοφορείς (Semaphores)

Προτάθηκαν από τον E. W. Dijkstra ως ένας

Νέος **τύπος** μεταβλητών της μορφής:

```
public class semaphore
{   private int s;
    // 0 και 1 αν είναι δυαδικοί (binary/boolean)
    // σηματοφορείς
    // στην υποπεριοχή 0..N των ακεραίων, N >1 αν
    // είναι γενικοί (general ή counting)

    // πράξεις (έστω ότι εκτελούνται από τη
    // διεργασία A)

    public wait()     [down / reserve / P (Prolagen)]

    { Αν s > 0, τότε s = s – 1 και η A
                  συνεχίζει την εκτέλεσή της;
      Αν s == 0, τότε η A εμποδίζεται πάνω στον
      s; // σταματά την εκτέλεσή της και
         // περιμένει, χωρίς να σπαταλά το χρόνο
         // της σε άσκοπες ανακυκλώσεις αναμονής
    }

    public signal (s)        [up / release / V (Verhogen)]

   {  Αν υπάρχουν διεργασίες εμποδισμένες πάνω
       στον s, τότε μία από αυτές ελευθερώνεται;
```

```
      Αν δεν υπάρχουν εμποδισμένες διεργασίες
        πάνω στον s, τότε η A συνεχίζει την
        εκτέλεσή της;
    }
} // semaphore
```

Σηματοφορείς

Οι πράξεις `wait` και `signal` είναι **αδιαίρετες** ή **ατομικές πράξεις (indivisible** ή **atomic operations),** δηλαδή:

- όταν μια διεργασία `A` αρχίσει μία από τις πράξεις `wait()` ή `signal()` πάνω σε κάποιο σηματοφορέα, καμία άλλη διεργασία δεν μπορεί να προσπελάσει το σηματοφορέα μέχρις ότου η `A`:
  - είτε εκτελέσει πλήρως την πράξη
  - είτε αναγκαστεί να περιμένει, επειδή ο σηματοφορέας έχει την τιμή `0`

ΔΙΑΦΟΡΟΙ ΟΡΙΣΜΟΙ (ποια διεργασία αφυπνίζεται)

- **Σύνολο εμποδισμένων (blocked set) διεργασιών πάνω στον `s`**
  - δεν προσδιορίζεται ποια διεργασία αφυπνίζεται όταν εκτελεστεί η εντολή `signal()`
  - πιθανότητα να υποσιτιστεί μια διεργασία, όταν το πλήθος των διεργασιών είναι μεγαλύτερο ή ίσο του `3`

- **Ουρά εμποδισμένων (blocked queue) διεργασιών στον `s`**
  - οι διεργασίες αφυπνίζονται με τη σειρά που έχουν εμποδιστεί (FIFO)
  - στην περίπτωση αυτή είναι αδύνατο να συμβεί υποσιτισμός

Σηματοφορείς

Παράδειγμα υλοποίησης σε UCSD Pascal:

- Προκαθορισμένος τύπος **`semaphore`**
- Τιμές στην περιοχή `0` ως **`maxint`**
- Ουρά εμποδισμένων (blocked queue) διεργασιών πάνω στον κάθε σηματοφορέα
- Προκαθορισμένες διαδικασίες:

  `procedure seminit (var s:semaphore; c: integer);`
  - δίνει την αρχική τιμή `c` στο σηματοφορέα `s`
  - δημιουργεί μία άδεια ουρά για τον `s`

  `procedure wait (var s: semaphore);`
  - κάνει τη διεργασία να περιμένει πάνω στον `s`

  `procedure signal (var s: semaphore);`
  - "σημαίνει" το σηματοφορέα `s`

  `procedure attach (s: semaphore; διάνυσμα: integer);`
  - συνδέει τον `s` με μια εξωτερική διακοπή (external interrupt) του συστήματος
  - Όταν συμβεί η διακοπή αυτή στο υλισμικό του υπολογιστή, τότε σημαίνεται ο σηματοφορέας `s`

Σηματοφορείς

Αμετάβλητες ιδιότητες ή αρχές (invariants) των σηματοφορέων:

αληθείς για κάθε κατάσταση κάθε δυνατής εκτέλεσης ενός παράλληλου προγράμματος

`s ≥ 0` **`s = s0 + #signal(s) - #wait(s)`**

όπου `s0` είναι η αρχική τιμή ενός σηματοφορέα `s` και
`#signal, #wait`: πλήθος των εντολών `signal(s)` και `wait(s)` *που έχουν εκτελεστεί πλήρως*
(απόδειξη: απευθείας από τον ορισμό)
Επίσης, από τις παραπάνω σχέσεις προκύπτει η σχέση:

`#wait(s) ≤ #signal(s) + s0`

Αν σ' ένα πρόγραμμα δεν ισχύει μία από τις σχέσεις αυτές, τότε το πρόγραμμα είναι λανθασμένο

Δυαδικοί Σηματοφορείς

Ανεξάρτητα από την υλοποίηση της πράξης `signal(s)` οι δυαδικοί σηματοφορείς μας επιτρέπουν την εύκολη λύση του προβλήματος του αμοιβαίου αποκλεισμού

```
process P1;
{
. . .
   while (true)
   {
    . . .
    wait (s);
    <κρίσιμη_
    περιοχή>;
    signal (s);
    . . .
    }
} // P1
```

```
process P2;
{
. . .
   while (true)
   {
    . . .
    wait (s);
    <κρίσιμη_
    περιοχή>;
    signal (s);
    . . .
    }
} // P2
```

```
process P3;
{
. . .
   while (true)
   {
    . . .
    wait (s);
    <κρίσιμη_
    περιοχή>;
    signal (s);
    . . .
    }
} // P3
```

Κύριο Πρόγραμμα:

```
{
 seminit (s, 1 {δυαδικός σηματοφορέας});
                                ⋮
 start (P1); start (P2); ... start (Pr);
                                ⋮
} // αμοιβαίος_αποκλεισμός
```

- **Απόδειξη**
  - αν `s0==1` => μόνο μία διεργασία μπορεί να εκτελέσει την εντολή `wait(s)` και να μπει στην ΚΠ της

- οι άλλες διεργασίες περιμένουν την εκτέλεση μιας εντολής `signal(s)`
- μία διεργασία εμποδίζεται μόνο αν κάποια άλλη είναι στην ΚΠ της, δηλαδή αν `s==0`

`=> #wait(s) = #signal(s) + 1`

Δυαδικοί Σηματοφορείς

Υποθέτουμε ότι η μεταβλητή γέφυρα_ελεύθερη έχει δηλωθεί ως τύπου semaphore

```
process T1;
{
   μη κρίσιμη_περιοχή ταξιδιού;
   wait (γέφυρα_ελεύθερη);
   κρίσιμη περιοχή χρήσης της γέφυρας;
   signal (γέφυρα_ελεύθερη);
} //T1;

process T2;
{
   μη κρίσιμη_περιοχή ταξιδιού;
   wait (γέφυρα_ελεύθερη);
   κρίσιμη περιοχή χρήσης της γέφυρας;
   signal (γέφυρα_ελεύθερη);
} // T2;

...

process Tn;
{
   μη κρίσιμη_περιοχή ταξιδιού;
   wait (γέφυρα_ελεύθερη);
   κρίσιμη περιοχή χρήσης της γέφυρας;
   signal (γέφυρα_ελεύθερη);
} // Tn;
```

Κύριο Πρόγραμμα:

```
{
```

```
  seminit (γέφυρα_ελεύθερη, 1);
  ... start (T1); start (T2); ... start (Tn); ...
} // προγραμματισμός_αμαξοστοιχιών
```

Συγχρονισμός διεργασιών (Process synchronization)

- συνεργαζόμενες ταυτόχρονες διεργασίες
- μία διεργασία περιμένει ένα *γεγονός (event)* από κάποια άλλη
- χρήση δυαδικών σηματοφορέων για να εμποδίζουμε και να ελευθερώνουμε διεργασίες κατά βούληση.

Παράδειγμα

Η `P1` θέλει σε κάποιο σημείο της εκτέλεσής της να ελευθερώσει την `P2`, που πρέπει να την περιμένει σε κάποιο άλλο σημείο της δικής της εκτέλεσης:

η μεταβλητή s έχει δηλωθεί ως τύπου semaphore

```
process P1;
{
    . . .
    // ξύπνα την P2
    signal (s);
    . . .
} // P1
```

```
process P2;
{
    . . .
    // περίμενε
    wait (s);
    . . .
} // P2
```

Κύριο Πρόγραμμα:

```
{
    seminit (s, 0);  start (P2);  start (P1);
} // ξύπνημα
```

Συγχρονισμός διεργασιών

- **Απόδειξη**
  - αν `s0==0` => η `P2` περιμένει, μέχρις ότου η `P1` εκτελέσει την εντολή `signal(s)`
  - Αν η `P1` εκτελέσει την `signal(s)` πριν η `P2` εκτελέσει την `wait(s)`, τότε η `P2` δεν περιμένει καθόλου

$$\Rightarrow \#wait(s) \leq \#signal(s)$$

- Ασύμμετρες (asymmetric) διεργασίες

  η `P2` ελέγχει την `P1` κι όχι αντίστροφα
- μερική περίπτωση του προβλήματος της διαδεργασιακής επικοινωνίας