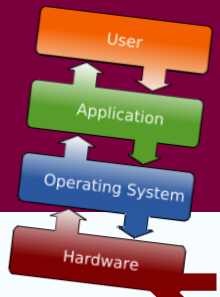


Συγχρονισμός:
- Αδιέξοδο
- Παρατεταμένη Στέρηση

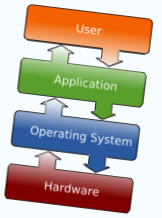
Ανδρέας Λ. Συμεωνίδης

Αν. Καθηγητής
Τμήμα Ηλεκτρολόγων Μηχ/κών
&
Μηχ/κών Υπολογιστών, Α.Π.Θ.

Email: asymeon@eng.auth.gr

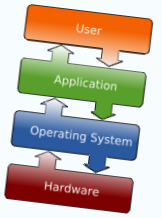
Στόχοι της Δ-6

- Να ολοκληρώσει τη μελέτη πάνω στην συνταύτιση
- Να ορίσει την έννοια του αδιεξόδου
- Να παρουσιάσει τις διαφορετικές τεχνικές για την πρόληψη, ανίχνευση και αποφυγή αδιεξόδων
- Να συζητήσει το πρόβλημα των συνδαιτυμόνων φιλοσόφων
- Να παρουσιάσει τη διαχείριση αδιεξόδων σε Windows, Linux και Solaris
- Να δώσει μια σειρά από πιθανές ασκήσεις πάνω στα αδιέξοδα

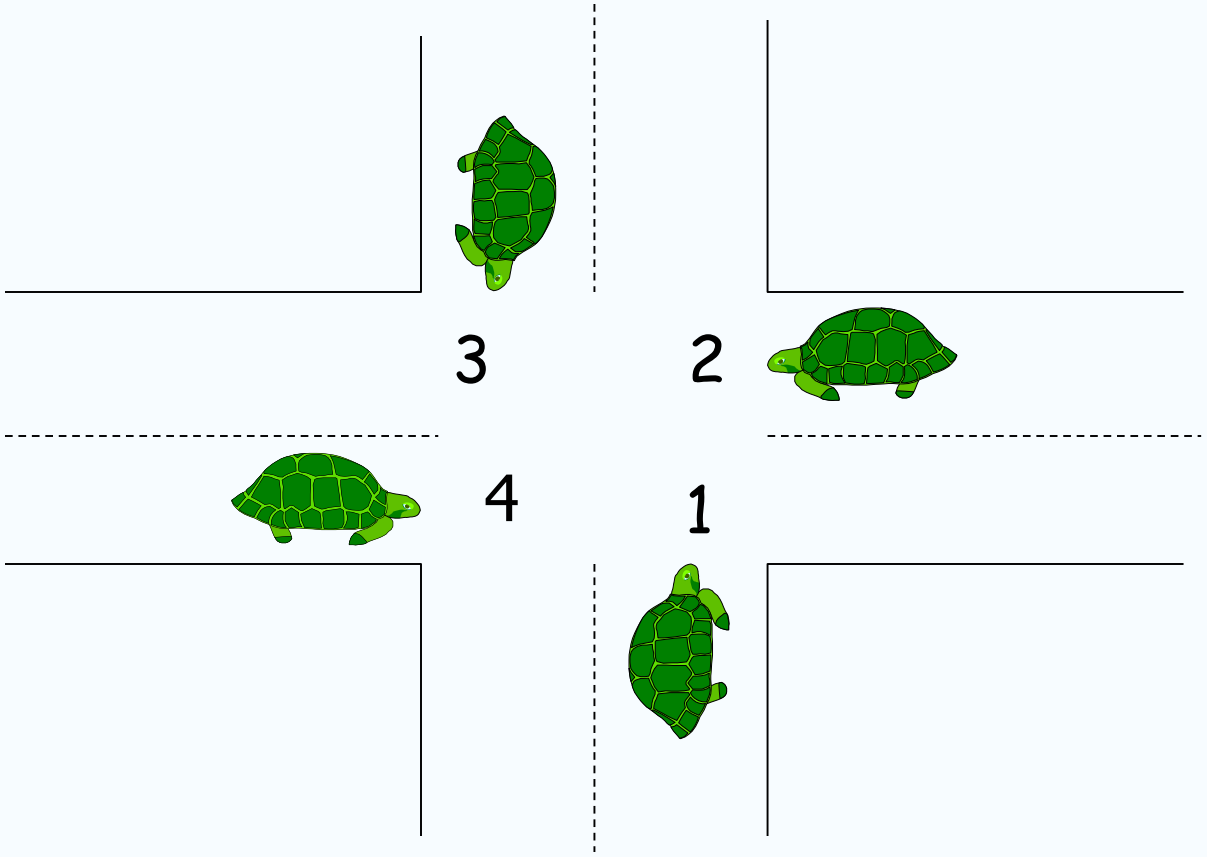


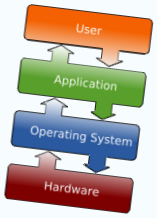
Αδιέξοδο

- Μόνιμη αναστολή ενός σετ από διεργασίες οι οποίες, είτε ανταγωνίζονται για κοινούς πόρους συστήματος, είτε επικοινωνούν μεταξύ τους
- Δεν υπάρχει ικανοποιητική λύση
- Εμπλέκουν αντικρουόμενες ανάγκες για πόρους από δυο ή περισσότερες διεργασίες

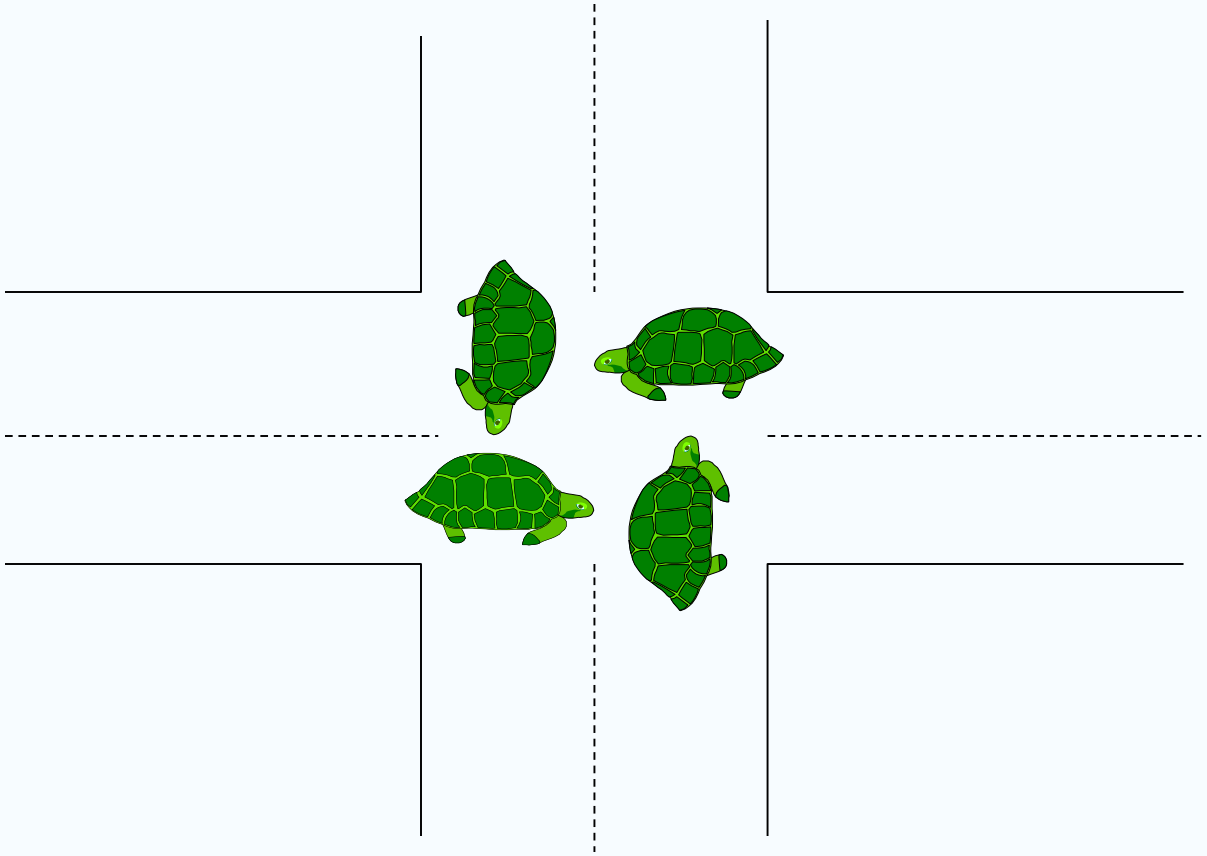


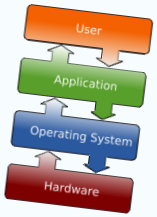
Το αδιέξοδο είναι πιθανό



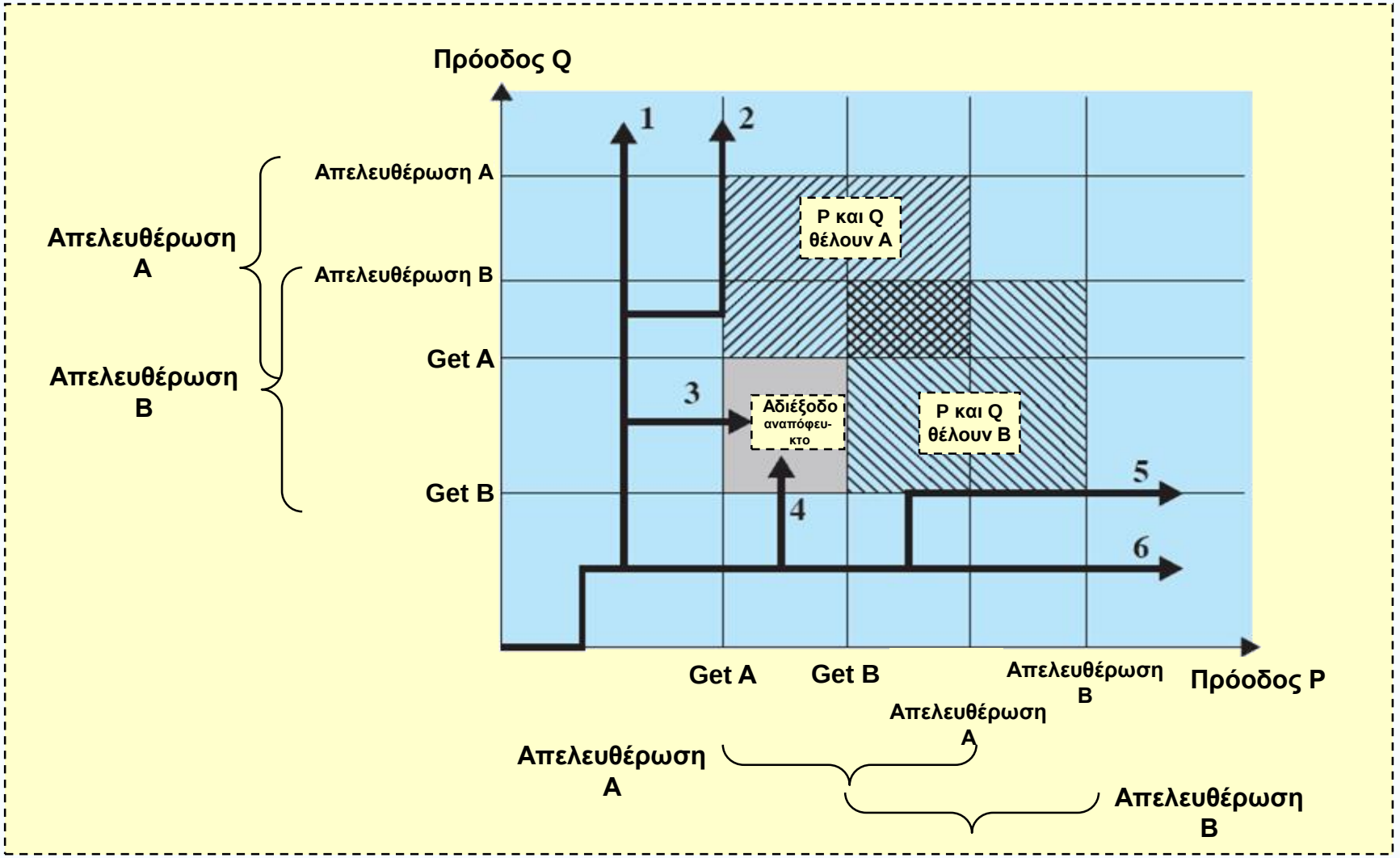


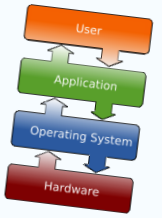
Αδιέξοδο



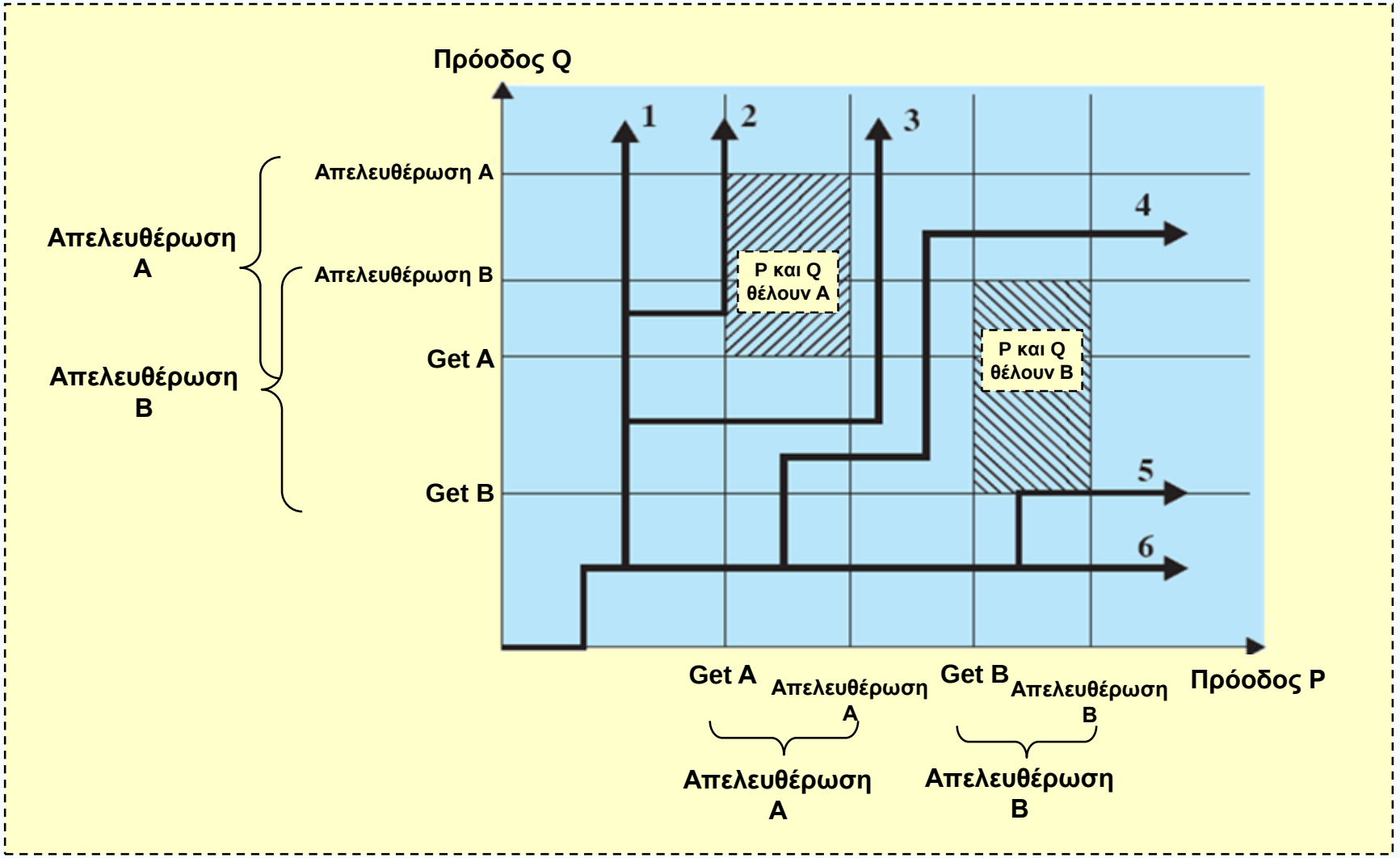


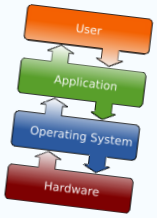
Αδιέξοδο


Πρόοδος Q
Απελευθέρωση A
Απελευθέρωση B
Get A
Get B
Απελευθέρωση A
Απελευθέρωση B
1
2
3
4
5
6
P και Q θέλουν A
Αδιέξοδο αναπόφευ-κτο
P και Q θέλουν B
Get A
Get B
Απελευθέρωση B
Πρόοδος P
Απελευθέρωση A
Απελευθέρωση A
Απελευθέρωση B


User
Application
Operating System
Hardware

Μη Αδιέξοδο

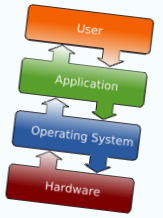

Πρόοδος Q
Απελευθέρωση A
Απελευθέρωση B
Get A
Get B
1
2
3
4
5
6
P και Q θέλουν A
P και Q θέλουν B
Απελευθέρωση A
Απελευθέρωση B
Get A
Get B
Απελευθέρωση A
Απελευθέρωση B
Πρόοδος P


User
Application
Operating System
Hardware

Πόροι (Resources)

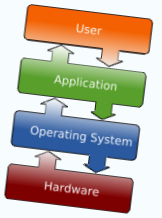
- **Προεκχωρούμενοι πόροι** (Preemptable Resources)
  - Μπορούν να απομακρυνθούν από μια διεργασία χωρίς παρενέργειες

- **Μη προεκχωρούμενοι πόροι** (Nonpreemptable Resources)
  - Προξενούν αποτυχία στη διεργασία όταν απομακρυνθούν



Επαναχρησιμοποιούμενοι πόροι

- Χρησιμοποιούνται μόνο από μια διεργασία τη φορά και δεν εξαντλούνται από τη χρήση αυτή
- Οι διεργασίες παίρνουν πόρους, τους οποίους ελευθερώνουν για επαναχρησιμοποίηση από άλλες διεργασίες
- Επεξεργαστές, κανάλια Ε/Ε, κύρια και δευτερεύουσα μνήμη, συσκευές και δομές δεδομένων όπως αρχεία, βάσεις δεδομένων και σηματοφόροι
- Το αδιέξοδο συμβαίνει εάν κάθε διεργασία κρατά κάποιον πόρο και αιτείται κάποιον άλλο



Επαναχρησιμοποιούμενοι πόροι (συν.)

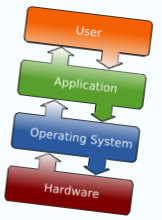
p0 p1 q0 q1 p2 q2....

Διεργασία P

Βήμα	Ενέργεια
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

Διεργασία Q

Βήμα	Ενέργεια
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)



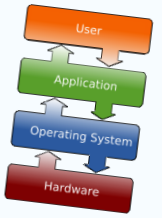
Επαναχρησιμοποιούμενοι πόροι (συν.)

- Υπάρχει διαθέσιμος χώρος για κατανομή 200Kbytes, και συμβαίνει η παρακάτω ακολουθία γεγονότων

```
          P1
 .  .  .
Request 80 Kbytes;
 .  .  .
Request 60 Kbytes;
```

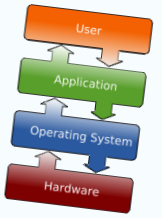
```
          P2
 .  .  .
Request 70 Kbytes;
 .  .  .
Request 80 Kbytes;
```

- Συμβαίνει αδιέξοδο αν και οι δυο διεργασίες περάσουν στο δεύτερο αίτημά τους



Αναλώσιμοι Πόροι

- Δημιουργούνται (παράγονται) και καταστρέφονται (καταναλώνονται)
- Διακοπές, σήματα, μηνύματα και πληροφορίες σε buffers Ε/Ε
- Μπορεί να χρειάζεται ένας σπάνιος συνδυασμός γεγονότων για να προκαλέσει αδιέξοδο



Παράδειγμα αδιεξόδου

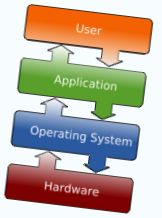
- Μπορεί να δημιουργηθεί αδιέξοδο αν υπάρχει μήνυμα **`Receive`** σε αναστολή

P1

```
...
Receive(P2);
...
Send(P2, M1);
```

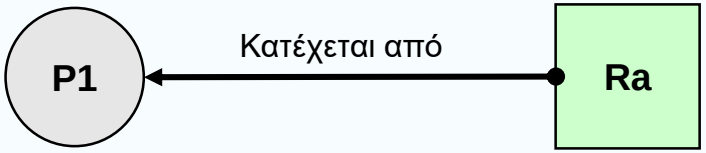
P2

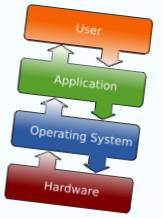
```
...
Receive(P1);
...
Send(P1, M2);
```



Γράφοι εκχώρισης πόρων

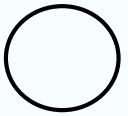
- Κατευθυνόμενοι γράφοι οι οποίοι απεικονίζουν μια κατάσταση των πόρων και των διεργασιών του συστήματος





Γράφοι εκχώρησης πόρων - Συμβολισμοί

- Διεργασία

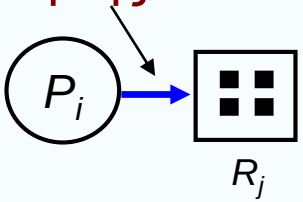


- Τύπος Πόρου με 4 στιγμιότυπα

Σειρά χρήσης πόρου μιας διεργασίας

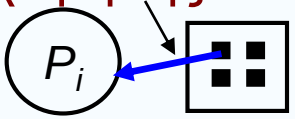
- P_i απαιτεί ένα στιγμιότυπο του R_j

Ακμή απαίτησης

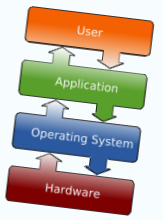


- P_i δεσμεύει ένα στιγμιότυπο του R_j

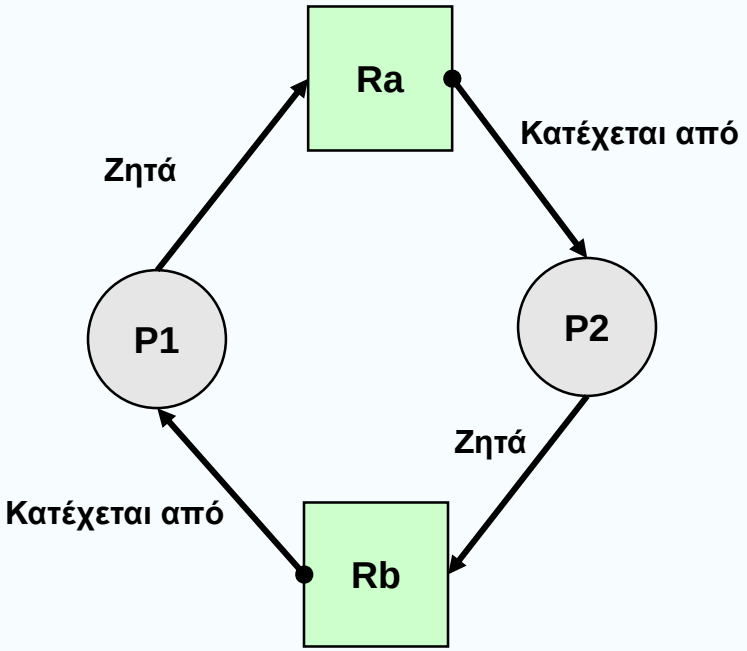
Ακμή εκχώρησης

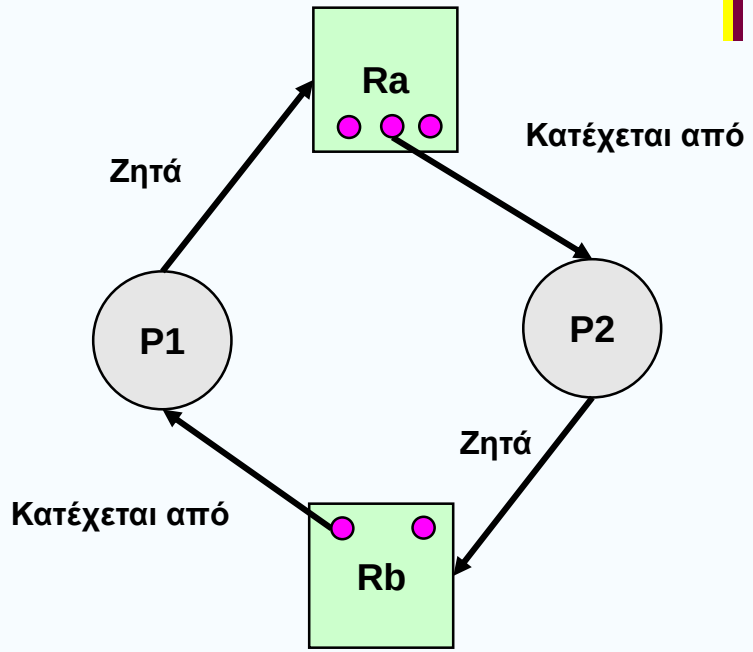


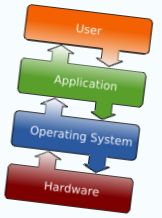
- P_i απελευθερώνει ένα στιγμιότυπο του R_j



Γράφοι εκχώρισης πόρων

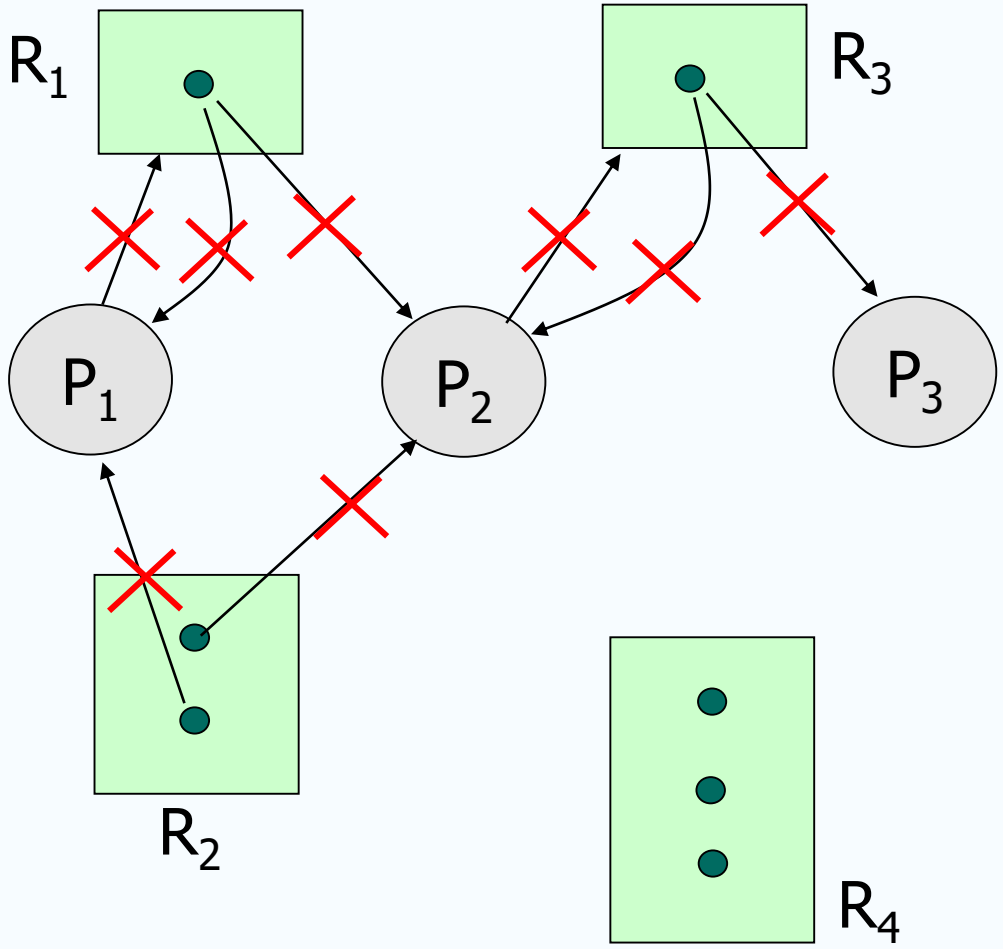


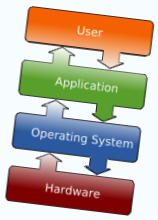




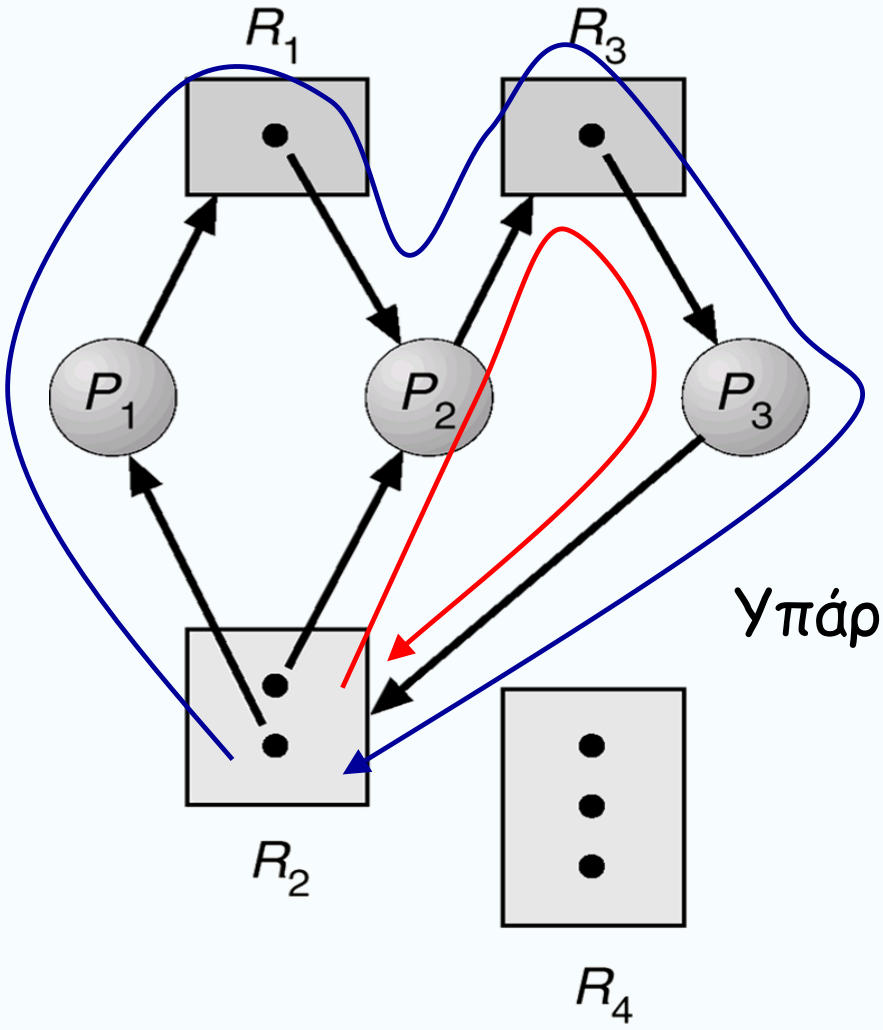
Παράδειγμα γράφου εκχώρησης πόρου

Μπορεί να συμβεί αδιέξοδο;

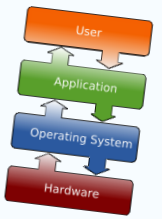




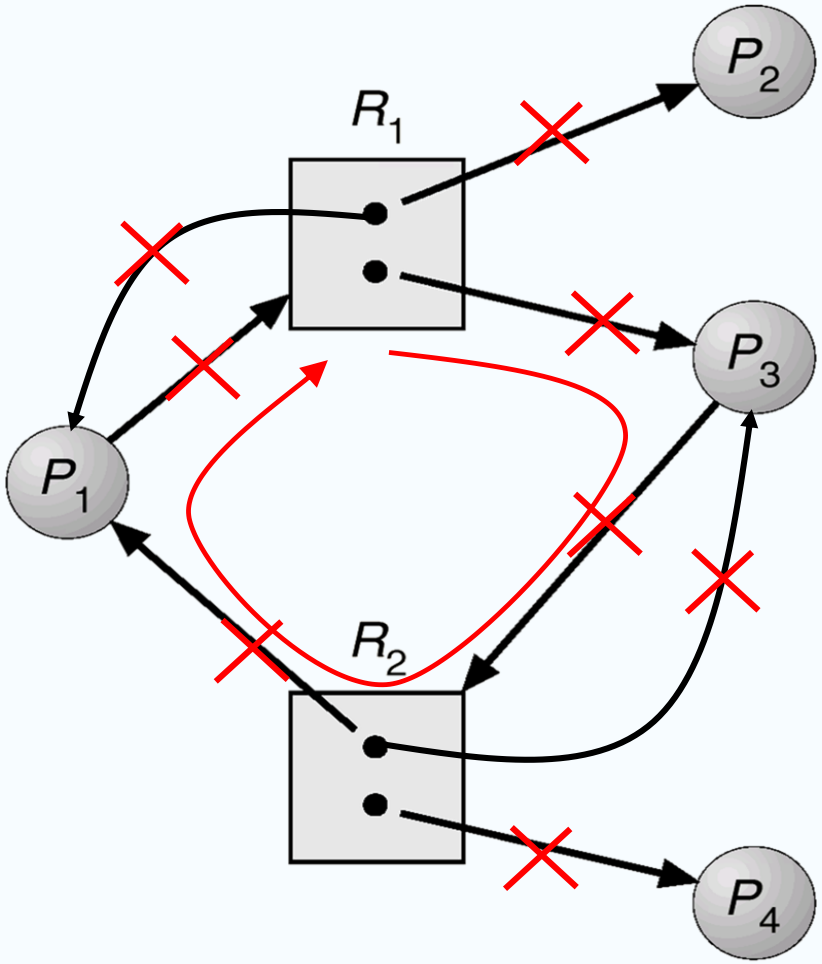
Παράδειγμα γράφου εκχώρησης πόρου με αδιέξοδο



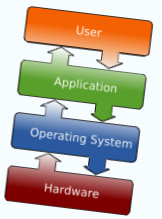
Υπάρχουν δύο κύκλοι



Παράδειγμα γράφου εκχώρησης πόρου με κύκλο αλλά χωρίς αδιέξοδο

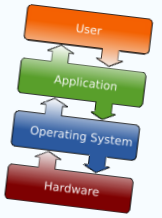


- Αν ο γράφος δεν περιέχει κύκλους ⇒ δεν υπάρχει αδιέξοδο
- Αν ο γράφος περιέχει ένα κύκλο ⇒
  - Αν υπάρχει μόνον ένα στιγμιότυπο ανά τύπο πόρου, τότε **υπάρχει αδιέξοδο**.
  - Αν υπάρχουν αρκετά στιγμιότυπα ανά τύπο πόρου, τότε **υπάρχει πιθανότητα να συμβεί αδιέξοδο**.



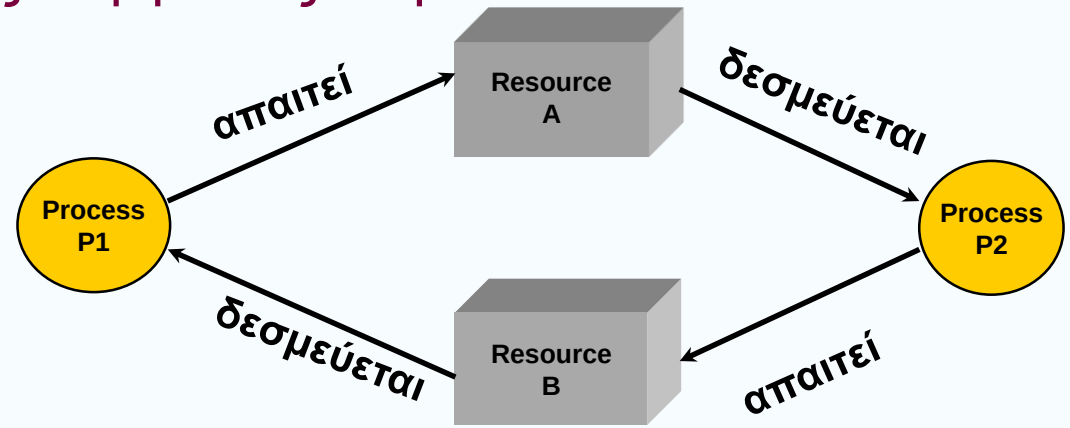
Συνθήκες αδιεξόδου

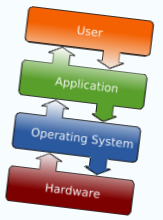
- Αμοιβαίος αποκλεισμός
  - Μόνο μια διεργασία μπορεί να χρησιμοποιήσει έναν πόρο τη φορά
- Κατοχή και Αναμονή (Hold-and-wait)
  - Μια διεργασία μπορεί να κρατά κατανεμημένους πόρους ενώ περιμένει ορισμό και άλλων



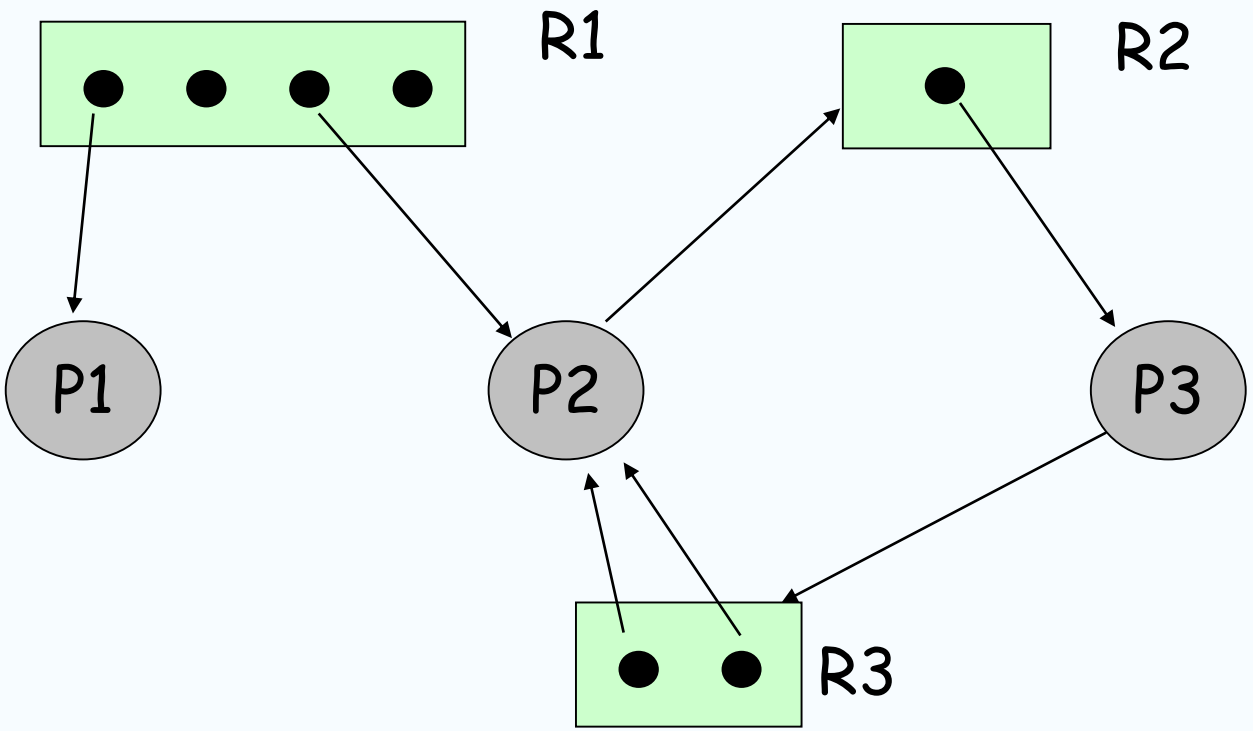
Συνθήκες αδιεξόδου (συν.)

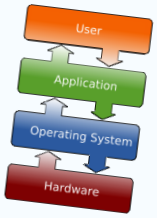
- Μη προεκχώρηση
  - Κανένας πόρος δεν μπορεί να αφαιρεθεί με τη βία από μια διεργασία που τον κατέχει
- Κυκλική αναμονή
  - Υπάρχει μια κλειστή αλυσίδα από διεργασίες, τέτοια ώστε κάθε διεργασία κατέχει τουλάχιστον έναν από τους αναγκαίους πόρους της επόμενης διεργασίας στην αλυσίδα



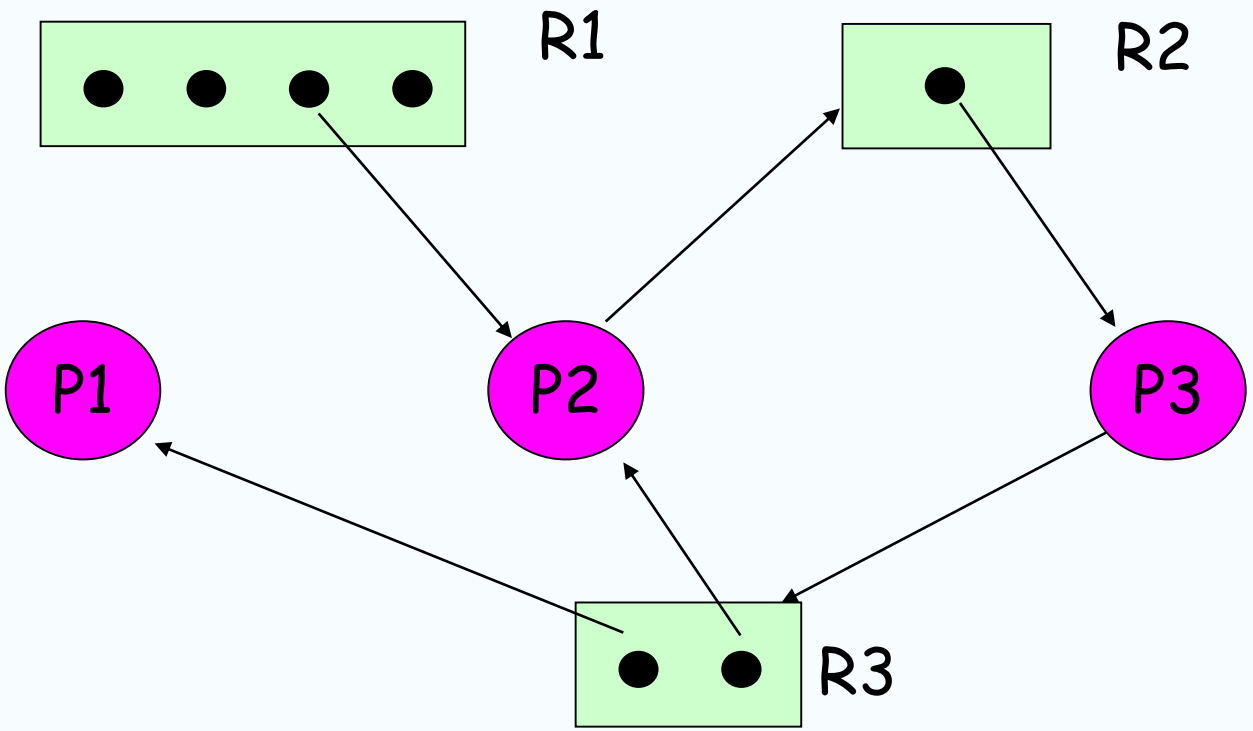


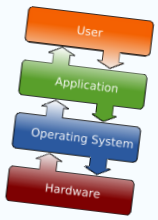
Παράδειγμα – κυκλική αναμονή





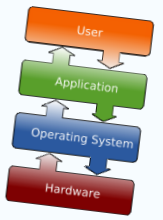
Παράδειγμα – δεν υπάρχει κυκλική αναμονή



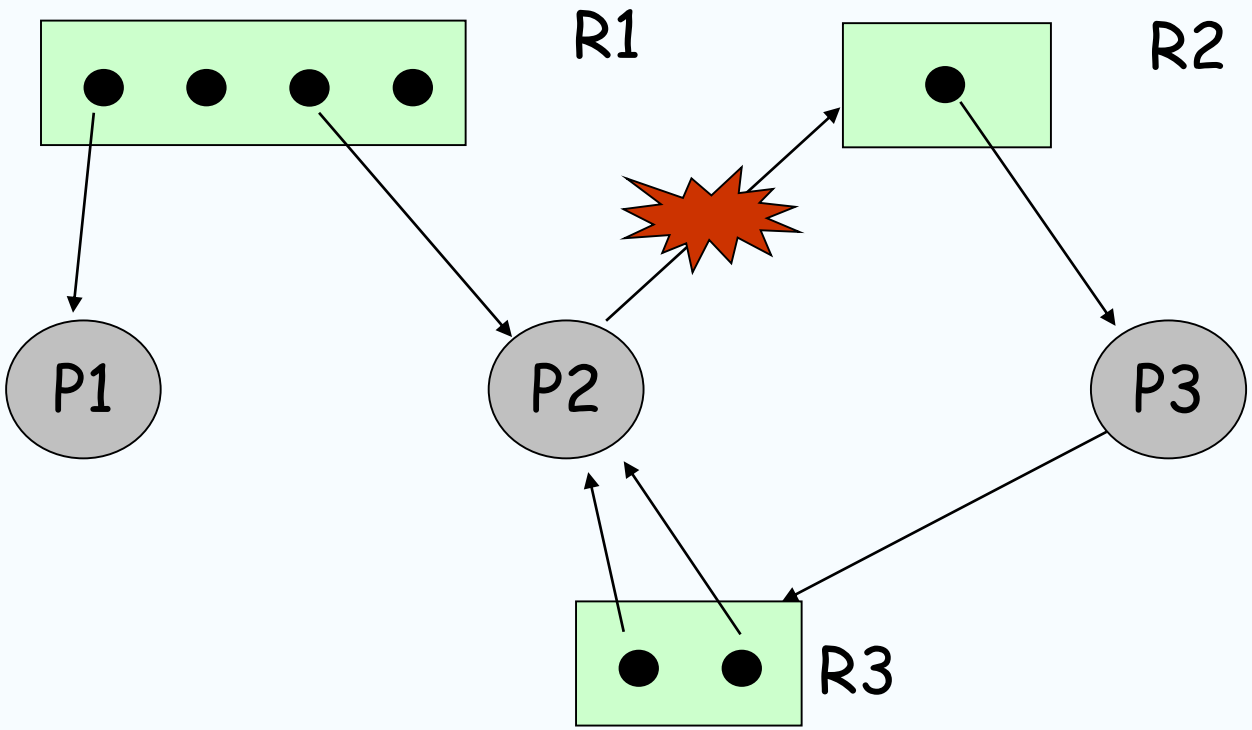


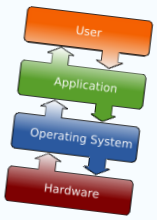
Πρόληψη αδιεξόδου

- Αμοιβαίος αποκλεισμός
  - Πρέπει να υποστηρίζεται από το ΛΣ
- Μη προεκχώρηση
  - Οι διεργασίες πρέπει να αποδεσμεύσουν τους πόρους και να αιτηθούν για αυτούς ξανά
  - Το ΛΣ μπορεί να προτιμήσει μια διεργασία για να απελευθερώσει τους πόρους του
- Κατοχή και αναμονή
  - Αίτηση της διεργασίας για όλους τους απαραίτητους πόρους σε μια φορά
- Κυκλική αναμονή
  - Καθορισμός μιας γραμμικής ταξινόμησης των τύπων πόρων



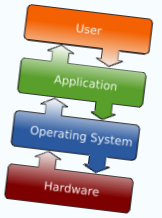
Πρόληψη κυκλικής αναμονής





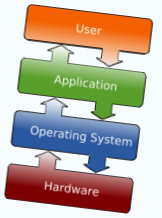
Αποφυγή αδιεξόδου

- Δυναμική λήψη απόφασης σχετικά με το κατά πόσο η τρέχουσα αίτηση δέσμευσης πόρων θα προκαλέσει, αν ικανοποιηθεί, αδιέξοδο
- Απαιτεί γνώση των μελλοντικών αιτήσεων για πόρους



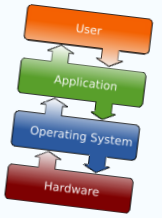
Δυο προσεγγίσεις για την αποφυγή αδιεξόδου

- Μην ξεκινάτε μια διεργασία της οποίας οι απαιτήσεις σε πόρους μπορεί να οδηγήσουν σε αδιέξοδο
- Μη χορηγείτε άδεια σε διεργασία με αυξανόμενες απαιτήσεις σε πόρους, αν η κατανομή σε πόρους μπορεί να οδηγήσει σε αδιέξοδο.



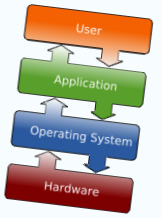
Αλγόριθμος του τραπεζίτη (Dijkstra 1965)

- Είναι γνωστός και ως ***άρνηση ανάθεσης πόρων***
- Δεν επιτρέπει την ικανοποίηση αυξημένων απαιτήσεων για πόρους σε μια διεργασία αν αυτή η εκχώρηση πόρων μπορεί να οδηγήσει σε αδιέξοδο.
- **Κατάσταση του συστήματος** : είναι η τρέχουσα ανάθεση πόρων στις διεργασίες
- **Ασφαλής κατάσταση** : υπάρχει μια τουλάχιστον ακολουθία εκτέλεσης των διεργασιών που μπορεί να εκτελεστεί μέχρι το τέλος (δηλαδή δεν οδηγεί σε αδιέξοδο)
- **Μη ασφαλής κατάσταση** : είναι η κατάσταση που δεν είναι ασφαλής



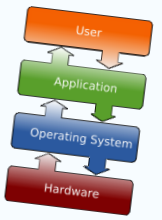
Ασφαλής κατάσταση

- Όταν μια διεργασία απαιτεί ένα διαθέσιμο πόρο, το σύστημα πρέπει να αποφασίσει αν η άμεση ανάθεση του πόρου θα το διατηρήσει σε ***ασφαλή κατάσταση***.
- Το σύστημα είναι σε ασφαλή κατάσταση αν υπάρχει μια ασφαλής ακολουθία για όλες τις διεργασίες.
- Η ακολουθία $\langle P_1, P_2, \ldots, P_n \rangle$ είναι ασφαλής αν για κάθε P_i, οι πόροι που η P_i μπορεί ακόμη να απαιτήσει μπορούν να ικανοποιηθούν από τους τρέχοντες διαθέσιμους πόρους συν τους πόρους που δεσμεύονται από όλες τις διεργασίες P_j, με $j < i$.



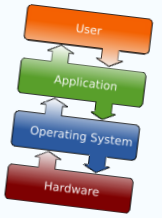
Ασφαλής κατάσταση (συν.)

- Αν οι ανάγκες σε πόρους της P_i δεν είναι άμεσα διαθέσιμοι, τότε η P_i μπορεί να περιμένει μέχρις ότου να τελειώσουν όλες οι P_j.
- Όταν τελειώνει η P_j, η P_i μπορεί να αποκτήσει τους πόρους που χρειάζεται, να εκτελεστεί, να επιστρέψει τους πόρους που της ανατέθηκαν και να τερματιστεί.
- Όταν τερματίσει η P_i, η P_{i+1} μπορεί να αποκτήσει τους πόρους που χρειάζεται κ.ο.κ.



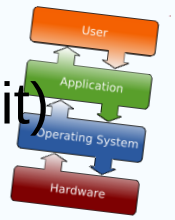
Βασικά γεγονότα

1. Αν το σύστημα είναι σε ασφαλή κατάσταση, δεν υπάρχει αδιέξοδο.
2. Αν το σύστημα δεν είναι σε ασφαλή κατάσταση, υπάρχει πιθανότητα αδιεξόδου



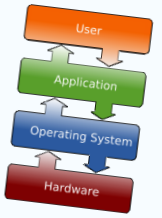
Παραδοχές

- Πολλαπλά στιγμιότυπα των πόρων
- Κάθε διεργασία πρέπει εκ των προτέρων να διεκδικεί τη μέγιστη χρήση των πόρων
- Όταν μια διεργασία απαιτεί έναν πόρο ίσως χρειαστεί να περιμένει
- Όταν μια διεργασία λάβει όλους τους πόρους πρέπει να τους επιστρέψει σε πεπερασμένο χρονικό διάστημα.
- Η μέγιστη απαίτηση για πόρους πρέπει να δηλώνεται εκ των προτέρων
- Οι σημαντικές διεργασίες πρέπει να είναι ανεξάρτητες και δεν υπόκεινται σε απαιτήσεις συγχρονισμού
- Πρέπει να υπάρχει ένας σταθερός αριθμός πόρων προς ανάθεση
- Καμιά διεργασία δεν μπορεί να περιέλθει σε κατάσταση εξόδου (exit) ενώ δεσμεύει πόρους



Δομές Δεδομένων για τον αλγόριθμο του τραπεζίτη

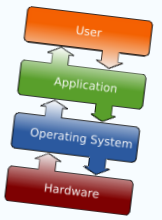
- ***n*** = το πλήθος των διεργασιών,
- ***m*** = το πλήθος τύπων πόρων
- ***Available***: Διάνυσμα μήκους ***m***. Αν ***available[j] = k***, υπάρχουν ***k*** στιγμιότυπα του τύπου πόρου R_j διαθέσιμα.
- ***Max:*** ***n x m*** πίνακας. Αν ***Max[i,j] = k***, τότε η διεργασία P_i μπορεί να απαιτήσει κατά μέγιστο ***k*** στιγμιότυπα του τύπου πόρου R_j.



Δομές Δεδομένων για τον αλγόριθμο του τραπεζίτη (συν.)

- ***Allocation:*** ***n x m*** πίνακας. Αν ***Allocation[i,j] = k*** τότε *στη διεργασία* P_i εκχωρούνται ***k*** στιγμιότυπα του πόρου R_j.
- ***Need:*** ***n x m*** πίνακας. Αν ***Need[i,j] = k***, τότε η διεργασία P_i μπορεί να χρειαστεί *k* επιπλέον στιγμιότυπα του πόρου R_j για να ολοκληρωθεί.

$$Need[i,j] = Max[i,j] - Allocation[i,j]$$



Καθορισμός μιας ασφαλούς κατάστασης: Αρχική κατάσταση

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Πίνακας Αιτήσεων C

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Πίνακας Κατανομής A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

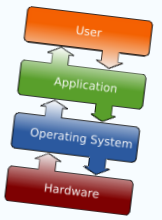
C – A

R1	R2	R3
9	3	6

Διάνυσμα Πόρων R

R1	R2	R3
0	0	1

Διαθέσιμο Διάνυσμα V



Καθορισμός μιας ασφαλούς κατάστασης: Η P_2 ολοκληρώνεται

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Πίνακας Αιτήσεων C

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Πίνακας Κατανομής A

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

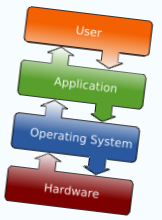
C – A

R1	R2	R3
9	3	6

Διάνυσμα Πόρων R

R1	R2	R3
6	2	3

Διαθέσιμο Διάνυσμα V



Καθορισμός μιας ασφαλούς κατάστασης: Η P_1 ολοκληρώνεται

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Πίνακας Αιτήσεων C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Πίνακας Κατανομής A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

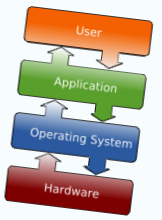
C – A

R1	R2	R3
9	3	6

Διάνυσμα Πόρων R

R1	R2	R3
7	2	3

Διαθέσιμο Διάνυσμα V



Καθορισμός μιας ασφαλούς κατάστασης: Η P_3 ολοκληρώνεται

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Πίνακας Αιτήσεων C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Πίνακας Κατανομής A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

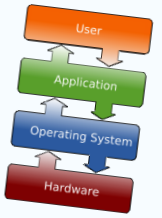
C – A

R1	R2	R3
9	3	6

Διάνυσμα Πόρων R

R1	R2	R3
9	3	4

Διαθέσιμο Διάνυσμα V



Καθορισμός μιας μη ασφαλούς κατάστασης: Αρχική κατάσταση

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Πίνακας Αιτήσεων C

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Πίνακας Κατανομής A

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

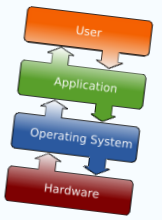
C – A

R1	R2	R3
9	3	6

Διάνυσμα Πόρων R

R1	R2	R3
1	1	2

Διαθέσιμο Διάνυσμα V



Καθορισμός μιας μη ασφαλούς κατάστασης: Η P_1 ζητά μια μονάδα από R_1 και R_3

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Πίνακας Αιτήσεων C

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Πίνακας Κατανομής A

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

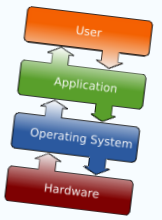
C – A

R1	R2	R3
9	3	6

Διάνυσμα Πόρων R

R1	R2	R3
0	0	1

Διαθέσιμο Διάνυσμα V



Παράδειγμα 2

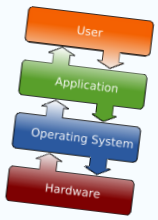
C = <8, 5, 9, 7>

Maximum Claim

Process	R_0	R_1	R_2	R_3
p_0	3	2	1	4
p_1	0	2	5	2
p_2	5	1	0	5
p_3	1	5	3	0
p_4	3	0	3	3

Allocated Resources

Process	R_0	R_1	R_2	R_3
p_0	2	0	1	1
p_1	0	1	2	1
p_2	4	0	0	3
p_3	0	2	1	0
p_4	1	0	3	0



Παράδειγμα 2 (συν.)

Maximum Claim

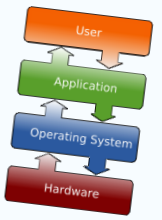
Process	R_0	R_1	R_2	R_3
p_0	3	2	1	4
p_1	0	2	5	2
p_2	5	1	0	5
p_3	1	5	3	0
p_4	3	0	3	3

Allocated Resources

Process	R_0	R_1	R_2	R_3
p_0	2	0	1	1
p_1	0	1	2	1
p_2	4	0	0	3
p_3	0	2	1	0
p_4	1	0	3	0
Sum	**7**	**3**	**7**	**5**

C = <8, 5, 9, 7>

- **Compute Total Held = <7,3,7,5>**



Παράδειγμα 2 (συν.)

Maximum Claim

Process	R_0	R_1	R_2	R_3
p_0	3	2	1	4
p_1	0	2	5	2
p_2	5	1	0	5
p_3	1	5	3	0
p_4	3	0	3	3

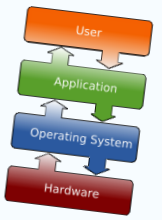
Allocated Resources

Process	R_0	R_1	R_2	R_3
p_0	2	0	1	1
p_1	0	1	2	1
p_2	4	0	0	3
p_3	0	2	1	0
p_4	1	0	3	0
Sum	**7**	**3**	**7**	**5**

C = <8, 5, 9, 7>

- **Compute Total Held = <7,3,7,5>**
- **Find Available Units (C-Held)**

avail = <8-7, 5-3, 9-7, 7-5>
= <1, 2, 2, 2>



Παράδειγμα 2 (συν.)

Maximum Claim

Process	R_0	R_1	R_2	R_3
p_0	3	2	1	4
p_1	0	2	5	2
p_2	5	1	0	5
p_3	1	5	3	0
p_4	3	0	3	3

Allocated Resources

Process	R_0	R_1	R_2	R_3
p_0	2	0	1	1
p_1	0	1	2	1
p_2	4	0	0	3
p_3	0	2	1	0
p_4	1	0	3	0
Sum	**7**	**3**	**7**	**5**

C = <8, 5, 9, 7>

- **Compute Total Held = <7,3,7,5>**
- **Find Available Units (C-Held)**

avail = <8-7, 5-3, 9-7, 7-5>
= <1, 2, 2, 2>

- **Can p0's maxc be met?**

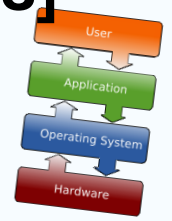
maxc[0,0]-alloc'[0,0] = 3-2 = 1≤1 = avail[0]

maxc[0,1]-alloc'[0,1] = 2-0 = 2≤2 = avail[1]

maxc[0,2]-alloc'[0,2] = 1-1 = 0≤2 = avail[2]

maxc[0,3]-alloc'[0,3] = 4-1 = 3≤2 ≠ avail[3]

- **Process p0 Fails on avail[3]**



Παράδειγμα 2 (συν.)

Maximum Claim

Process	R_0	R_1	R_2	R_3
p_0	3	2	1	4
p_1	0	2	5	2
p_2	5	1	0	5
p_3	1	5	3	0
p_4	3	0	3	3

Allocated Resources

Process	R_0	R_1	R_2	R_3
p_0	2	0	1	1
p_1	0	1	2	1
p_2	4	0	0	3
p_3	0	2	1	0
p_4	1	0	3	0
Sum	**7**	**3**	**7**	**5**

C = <8, 5, 9, 7>

- Compute Total Held = <7,3,7,5>
- Find Available Units (C-Held)

avail = <8-7, 5-3, 9-7, 7-5>
= <1, 2, 2, 2>

- Can p1's maxc be met?

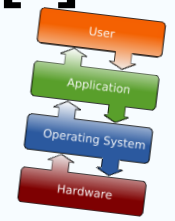
maxc[1,0]-alloc'[1,0] = 0-0 = 0≤1 = avail[0]

maxc[1,1]-alloc'[1,1] = 2-1 = 1≤2 = avail[1]

maxc[1,2]-alloc'[1,2] = 5-2 = 3≤2 = avail[2]

maxc[1,3]-alloc'[1,3] = 2-1 = 1≤2 = avail[3]

- Process p1 Fails on avail[2]



Παράδειγμα 2 (συν.)

Maximum Claim

Process	R_0	R_1	R_2	R_3
p_0	3	2	1	4
p_1	0	2	5	2
p_2	5	1	0	5
p_3	1	5	3	0
p_4	3	0	3	3

Allocated Resources

Process	R_0	R_1	R_2	R_3
p_0	2	0	1	1
p_1	0	1	2	1
p_2	4	0	0	3
p_3	0	2	1	0
p_4	1	0	3	0
Sum	**7**	**3**	**7**	**5**

C = <8, 5, 9, 7>

- **Compute Total Held = <7,3,7,5>**
- **Find Available Units (C-Held)**

avail = <8-7, 5-3, 9-7, 7-5>
= <1, 2, 2, 2>

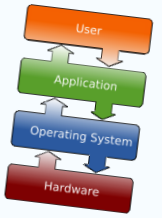
- **Can p2's maxc be met?**

maxc[2,0]-alloc'[2,0] = 5-4 = 1≤1 = avail[0]

maxc[2,1]-alloc'[2,1] = 1-0 = 1≤2 = avail[1]

maxc[2,2]-alloc'[2,2] = 0-0 = 0≤2 = avail[2]

maxc[2,3]-alloc'[2,3] = 5-3 = 2≤2 = avail[3]



Παράδειγμα 2 (συν.)

Maximum Claim

Process	R_0	R_1	R_2	R_3
p_0	3	2	1	4
p_1	0	2	5	2
p_2	5	1	0	5
p_3	1	5	3	0
p_4	3	0	3	3

Allocated Resources

Process	R_0	R_1	R_2	R_3
p_0	2	0	1	1
p_1	0	1	2	1
p_2	4	0	0	3
p_3	0	2	1	0
p_4	1	0	3	0
Sum	**7**	**3**	**7**	**5**

C = <8, 5, 9, 7>

- **Compute Total Held = <7,3,7,5>**
- **Find Available Units (C-Held)**

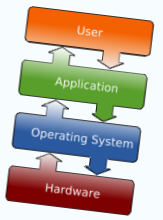
avail = <8-7, 5-3, 9-7, 7-5>
= <1, 2, 2, 2>

- **Can p2's maxc be met?**

maxc[2,0]-alloc'[2,0] = 5-4 = 1≤1 = avail[0]
maxc[2,1]-alloc'[2,1] = 1-0 = 1≤2 = avail[1]
maxc[2,2]-alloc'[2,2] = 0-0 = 0≤2 = avail[2]
maxc[2,3]-alloc'[2,3] = 5-3 = 2≤2 = avail[3]

- **Yes! Redo avail/Update alloc'**

avail[0] = avail[0]+alloc'[2,0] = 1+4 = 5
avail[1] = avail[1]+alloc'[2,1] = 2+0 = 2
avail[2] = avail[2]+alloc'[2,2] = 2+0 = 2
avail[3] = avail[3]+alloc'[2,3] = 2+3 = 5



Παράδειγμα 2 (συν.)

Maximum Claim

Process	R_0	R_1	R_2	R_3
p_0	3	2	1	4
p_1	0	2	5	2
p_2	5	1	0	5
p_3	1	5	3	0
p_4	3	0	3	3

Allocated Resources

Process	R_0	R_1	R_2	R_3
p_0	2	0	1	1
p_1	0	1	2	1
p_2	0	0	0	0
p_3	0	2	1	0
p_4	1	0	3	0
Sum	3	3	7	2

C = <8, 5, 9, 7>

- Recompute Total Held=<3,3,7,2>
- Find Available Units (C-Held)

avail = <8-3, 5-3, 9-7, 7-2>
= <5, 2, 2, 5>

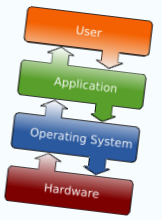
- Can anyone's maxc be met?

maxc[4,0]-alloc'[4,0] = 5-1 = 4≤5 = avail[0]

maxc[4,1]-alloc'[4,1] = 0-0 = 0≤2 = avail[1]

maxc[4,2]-alloc'[4,2] = 3-3 = 0≤2 = avail[2]

maxc[4,3]-alloc'[4,3] = 3-0 = 3≤5 = avail[3]



Παράδειγμα 2 (συν.)

Maximum Claim

Process	R_0	R_1	R_2	R_3
p_0	3	2	1	4
p_1	0	2	5	2
p_2	5	1	0	5
p_3	1	5	3	0
p_4	3	0	3	3

Allocated Resources

Process	R_0	R_1	R_2	R_3
p_0	2	0	1	1
p_1	0	1	2	1
p_2	0	0	0	0
p_3	0	2	1	0
p_4	1	0	3	0
Sum	**3**	**3**	**7**	**2**

C = <8, 5, 9, 7>

- Recompute Total Held=<3,3,7,2>
- Find Available Units (C-Held)

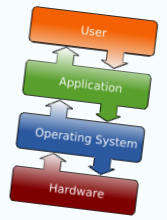
avail = <8-3, 5-3, 9-7, 7-2>
= <5, 2, 2, 5>

- Can anyone's maxc be met?

maxc[4,0]-alloc'[4,0] = 5-1 = 4≤5 = avail[0]
maxc[4,1]-alloc'[4,1] = 0-0 = 0≤2 = avail[1]
maxc[4,2]-alloc'[4,2] = 3-3 = 0≤2 = avail[2]
maxc[4,3]-alloc'[4,3] = 3-0 = 3≤5 = avail[3]

- P_4 can exercise max claim

avail[0] = avail[0]+alloc'[4,0] = 5+1 = 6
avail[1] = avail[1]+alloc'[4,1] = 2+0 = 2
avail[2] = avail[2]+alloc'[4,2] = 2+3 = 5
avail[3] = avail[3]+alloc'[4,3] = 5+0 = 5



Παράδειγμα 2 (συν.)

Maximum Claim

Process	R_0	R_1	R_2	R_3
p_0	3	2	1	4
p_1	0	2	5	2
p_2	5	1	0	5
p_3	1	5	3	0
p_4	3	0	3	3

Allocated Resources

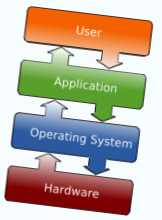
Process	R_0	R_1	R_2	R_3
p_0	2	0	1	1
p_1	0	1	2	1
p_2	0	0	0	0
p_3	0	2	1	0
p_4	0	0	0	0
Sum	**2**	**3**	**4**	**2**

C = <8, 5, 9, 7>

- Recompute Total Held=<2,3,4,2>
- Find Available Units (C-Held)

avail = <8-2, 5-3, 9-4, 7-2>
= <6, 2, 5, 5>

- Can anyone's maxc be met?
- Yes, any of them can!
- Choose p0



Παράδειγμα 2 (συν.)

Maximum Claim

Process	R_0	R_1	R_2	R_3
p_0	3	2	1	4
p_1	0	2	5	2
p_2	5	1	0	5
p_3	1	5	3	0
p_4	3	0	3	3

Allocated Resources

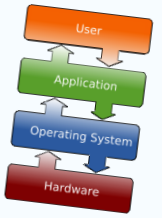
Process	R_0	R_1	R_2	R_3
p_0	0	0	0	0
p_1	0	1	2	1
p_2	0	0	0	0
p_3	0	2	1	0
p_4	0	0	0	0
Sum	**0**	**3**	**4**	**2**

C = <8, 5, 9, 7>

- Recompute Total Held=<0,3,4,2>
- Find Available Units (C-Held)

avail = <8-0, 5-3, 9-4, 7-2>
= <8, 2, 5, 5>

- Can anyone's maxc be met?
- Yes, any of them can!
- Choose p1



Παράδειγμα 2 (συν.)

Maximum Claim

Process	R_0	R_1	R_2	R_3
p_0	3	2	1	4
p_1	0	2	5	2
p_2	5	1	0	5
p_3	1	5	3	0
p_4	3	0	3	3

Allocated Resources

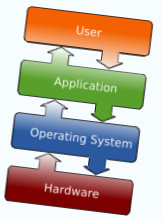
Process	R_0	R_1	R_2	R_3
p_0	0	0	0	0
p_1	0	0	0	0
p_2	0	0	0	0
p_3	0	2	1	0
p_4	0	0	0	0
Sum	**0**	**2**	**1**	**0**

C = <8, 5, 9, 7>

- Recompute Total Held=<0,2,1,0>
- Find Available Units (C-Held)

avail = <8-0, 5-2, 9-1, 7-0>
= <8, 3, 8, 7>

- Can anyone's maxc be met?
- Yes, Choose p3



Παράδειγμα 2: περίληψη

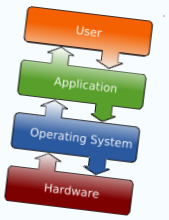
Maximum Claim

Process	R_0	R_1	R_2	R_3
p_0	3	2	1	4
p_1	0	2	5	2
p_2	5	1	0	5
p_3	1	5	3	0
p_4	3	0	3	3

Allocated Resources

Process	R_0	R_1	R_2	R_3
p_0	0	0	0	0
p_1	0	0	0	0
p_2	0	0	0	0
p_3	0	0	0	0
p_4	0	0	0	0
Sum	0	0	0	0

- Ο Alloc' μηδενίζεται
- Ασφαλής κατάσταση με ολοκληρωμένες όλες τις διεργασίες
- Τερματίζει ο αλγόριθμος με επιτυχία
- Διεκπεραίωση διεργασιών με την ακόλουθη σειρά:
  - **p2, p4, p0, p1, p3**
  - Ασφαλής Εκτέλεση
- **Όλες οι μέγιστες απαιτήσεις ικανοποιήθηκαν με κάποια σειρά**
- **Δεν υπήρξε αδιέξοδο ή μη ασφαλής κατάσταση**



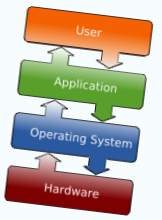
Λογική αποφυγής αδιεξόδων

Δομές Ολικών Δεδομένων

```
struct state {
        int resource[m];
        int available[m];
        int claim[n][m];
        int alloc[n][m];
}
```

Αλγόριθμος πόρου alloc

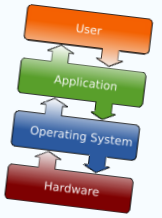
```
if (alloc [i,*] + request [*] > claim [i,*])
     < error >;                                  /* total request > claim*/
else if (request [*] > available [*])
     < suspend process >;
else {                                                /* simulate alloc */
     < define newstate by:
     alloc [i,*] = alloc [i,*] + request [*];
     available [*] = available [*] - request [*] >;
}
if (safe (newstate))
     < carry out allocation >;
else {
     < restore original state >;
     < suspend process >;
}
```



Λογική αποφυγής αδιεξόδων (συν.)

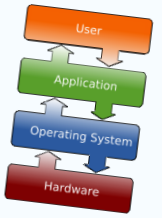
Έλεγχος για τον αλγόριθμο ασφαλείας (αλγόριθμος Τραπεζίτη)

```
boolean safe (state S) {
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible) {
        <find a process Pk in rest such that
            claim [k,*] – alloc [k,*] <= currentavail;>
        if (found) {                          /* simulate execution of Pk */
            currentavail = currentavail + alloc [k,*];
            rest = rest - {Pk};
        }
        else possible = false;
    }
    return (rest == null);
}
```



Αποφυγή αδιεξόδων

- Η μέγιστη απαίτηση για πόρους πρέπει να δηλωθεί από την αρχή
- Οι υποψήφιες διεργασίες πρέπει να είναι ανεξάρτητες (χωρίς απαιτήσεις συγχρονισμού)
- Πρέπει να υπάρχει ένας συγκεκριμένος αριθμός πόρων για κατανομή
- Καμιά διεργασία δεν μπορεί να φύγει (exit), ενώ κατέχει πόρους



Ανίχνευση αδιεξόδων

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Πίνακας Αιτήσεων Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

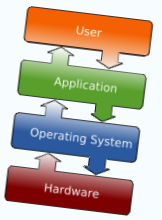
Πίνακας Κατανομής A

R1	R2	R3	R4	R5
2	1	1	2	1

Διάνυσμα Πόρων R

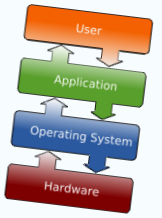
R1	R2	R3	R4	R5
0	0	0	0	1

Διαθέσιμο Διάνυσμα V



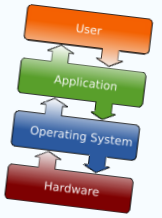
Στρατηγικές εφόσον ανιχνευτεί αδιέξοδο

- Εγκατάλειψη όλων των διεργασιών σε αδιέξοδο
- Επαναφορά και αποθήκευση κάθε διεργασίας σε αδιέξοδο σε κάποιο προηγούμενο σημείο ελέγχου (checkpoint) και επανεκκίνηση όλων των διεργασιών
  - Μπορεί να προκύψει ένα αρχικό αδιέξοδο
- Ακολουθιακά, εγκατάλειψη όλων των διεργασιών σε αδιέξοδο, μέχρι να πάψει το αδιέξοδο
- Ακολουθιακά, προεκχώρηση των πόρων, μέχρι να πάψει το αδιέξοδο



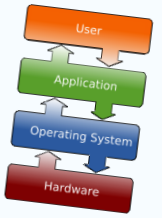
Πλεονεκτήματα και Μειονεκτήματα

Προσέγγιση	Πολιτική Κατανομής Πόρων	Παραλλαγές	Κύρια Πλεονεκτήματα	Κύρια Μειονεκτήματα
Πρόληψη	Συντηρητική: υποδεσμεύει πόρους	Αίτηση για όλους τους πόρους την ίδια στιγμή	- Κατάλληλη για διεργασίες που εκτελούν μεγάλο αριθμό δραστηριοτήτων - Η προεκχώρηση δεν είναι απαραίτητη	- Ανεπαρκής - Καθυστερεί την έναρξη των διεργασιών - Οι μελλοντικές απαιτήσεις σε πόρους πρέπει να είναι γνωστές
		Προεκχώρηση	- Κατάλληλη στις περιπτώσεις που η κατάσταση μπορεί να αποθηκεύεται και να ανακτάται με ευκολία	- Προεκχωρεί πόρους πιο συχνά απ' ότι θα έπρεπε - Υπόκειται σε κυκλικές επανενάρξεις
		Διάταξ πόρων	- Πραγματοποιείται κατά τη διάρκεια ελέγχων μεταγλώττισης - Δεν απαιτεί δυναμικούς υπολογισμούς	- Προεκχωρεί χωρίς μεγάλη χρήση - Δεν επιτρέπει την αύξηση των απαιτήσεων σε πόρους

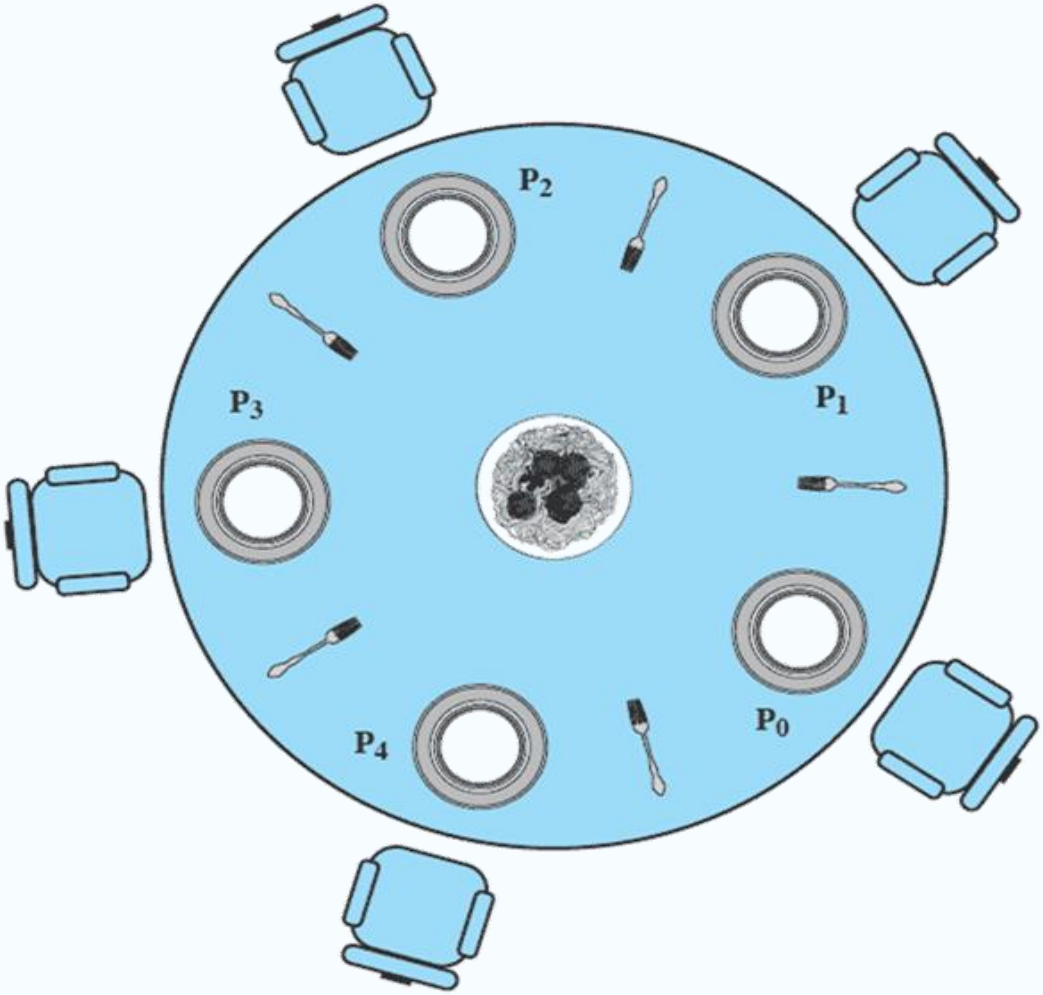


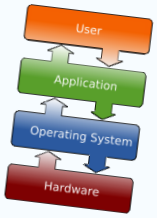
Πλεονεκτήματα και Μειονεκτήματα (συν.)

Προσέγγιση	Πολιτική Κατανομής Πόρων	Παραλλαγές	Κύρια Πλεονεκτήματα	Κύρια Μειονεκτήματα
Αποφυγή	Ενδιάμεση κατάσταση	Έξυπνος χειρισμός για την ανέυρεση μιας τουλάχιστον ασφαλούς διαδρομής	- Η προεκχώρηση δεν είναι απαραίτητη	- Οι μελλοντικές απαιτήσεις σε πόρους πρέπει να είναι γνωστές - Οι διεργασίες μπορεί να ανασταλούν για μεγάλες περιόδους
Ανίχνευση	Σε αφθονία: οι απαιτούμενοι πόροι παρέχονται όπου είναι δυνατόν	Περιοδικός έλεγχος αδιεξόδου	- Δεν καθυστερεί ποτέ την έναρξη των διεργασιών - Διευκολύνει τον online χειρισμό	- Έμφυτες απώλειες προεκχώρησης



Το Πρόβλημα του "Δείπνου των Φιλοσόφων"

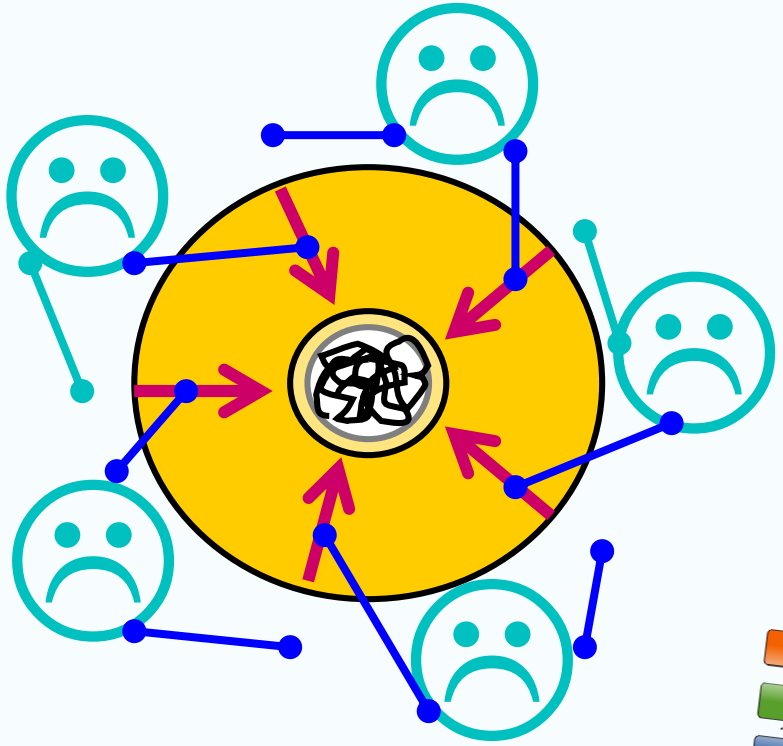


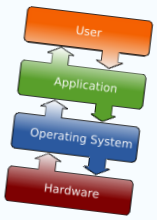


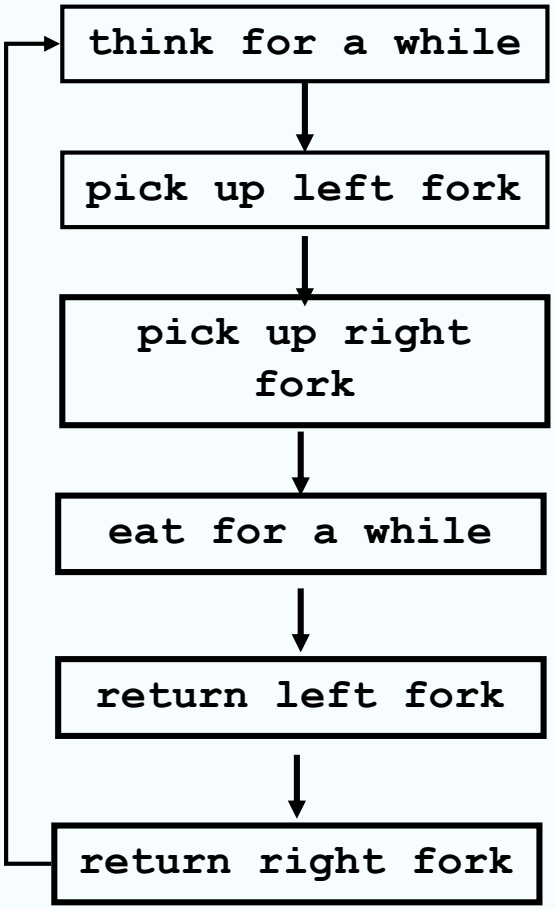
Η προφανής λύση

```
think for a while
pick up left fork
pick up right fork
eat for a while
return left fork
return right fork
```

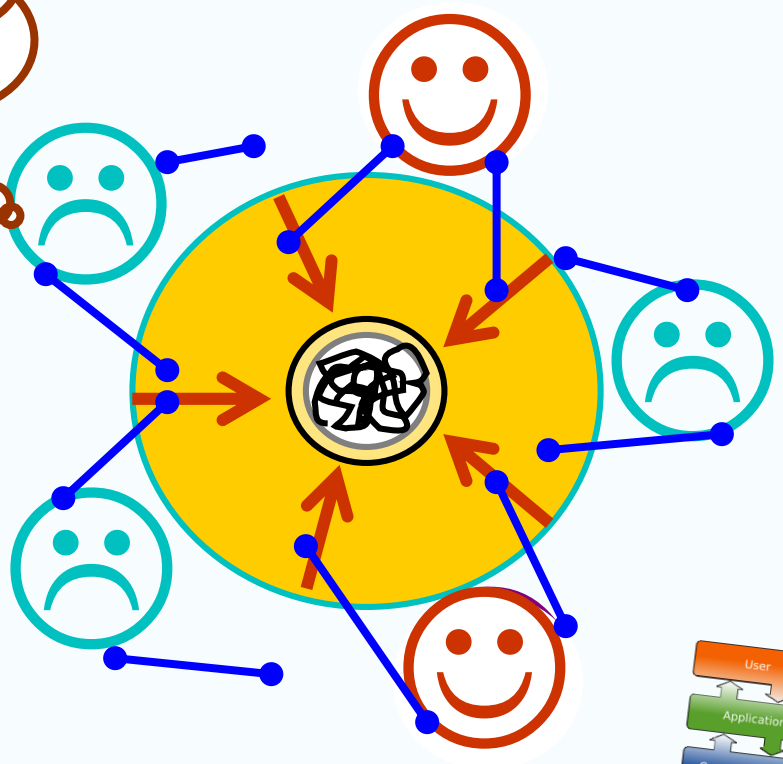
Μπορεί να συμβεί αδιέξοδο…

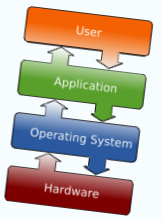






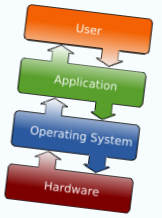
Η παρατεταμένη στέρηση είναι πρόβλημα...





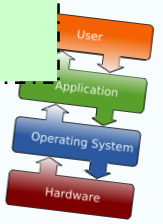
Λύσεις...

- Δεν υπάρχει συμμετρική λύση
- **Λύσεις:**
  - Προσθήκη ενός ακόμη πηρουνιού
  - Μέγιστος αριθμός 4 φιλοσόφων στο τραπέζι (κυκλική αναμονή)
  - Να εκτελείται διαφορετική αλληλουχία ενεργειών για τους φιλοσόφους με άρτιο και περιττό αύξοντα αριθμό δηλαδή δημιουργία δύο ομάδων φιλοσόφων.
    - Η μία ομάδα (περιττός α/α) θα αποκτά πρώτα το δεξιό και μετά το αριστερό πηρούνι και η άλλη ομάδα (άρτιος α/α) πρώτα το αριστερό και μετά το δεξιό πηρούνι.
  - Ένας φιλόσοφος επιτρέπεται να αποκτήσει τα πηρούνια μόνον όταν και τα δύο είναι διαθέσιμα (κρίσιμο τμήμα) - (κατοχή και αναμονή)
  - Σχεδιασμός του συστήματος έτσι ώστε ένας φιλόσοφος να «κλέψει» ένα πηρούνι που δεν είναι γειτονικό του.
  - Αλγόριθμος Lehmann-Rabin (non deterministic)



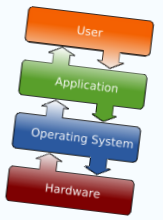
Το Πρόβλημα του "Δείπνου των Φιλοσόφων": Λύση 1

```
/* program      diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true) {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1), philosopher
(2),
          philosopher (3), philosopher (4));
     }
```



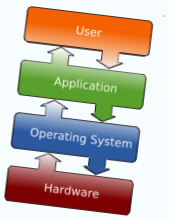
Το Πρόβλημα του "Δείπνου των Φιλοσόφων": Λύση 2

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
      think();
      wait (room);
      wait (fork[i]);
      wait (fork [(i+1) mod 5]);
      eat();
      signal (fork [(i+1) mod 5]);
      signal (fork[i]);
      signal (room);
    }

}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
}
```



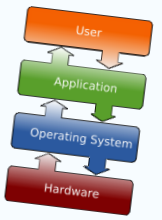
Το Πρόβλημα του "Δείπνου των Φιλοσόφων"

```
monitor dining_controller;
cond ForkReady[5];          /* condition variable for synchronization */
boolean fork[5] = {true};       /* availability status of each fork */

void get_forks(int pid)         /* pid is the philosopher id number */
{
   int left = pid;
   int right = (++pid) % 5;
   /*grant the left fork*/
   if (!fork(left)
      cwait(ForkReady[left]);          /* queue on condition variable */
   fork(left) = false;
   /*grant the right fork*/
   if (!fork(right)
      cwait(ForkReady(right);          /* queue on condition variable */
   fork(right) = false:
}
void release_forks(int pid)
{
   int left = pid;
   int right = (++pid) % 5;
   /*release the left fork*/
   if (empty(ForkReady[left])      /*no one is waiting for this fork */
      fork(left) = true;
   else                          /* awaken a process waiting on this fork */
      csignal(ForkReady[left]);
   /*release the right fork*/
   if (empty(ForkReady[right])     /*no one is waiting for this fork */
      fork(right) = true;
   else                          /* awaken a process waiting on this fork */
      csignal(ForkReady[right]);
}
```



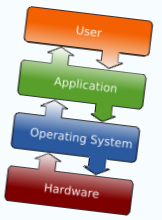
Το Πρόβλημα του "Δείπνου των Φιλοσόφων"

```
void philosopher[k=0 to 4]              /* the five philosopher clients */
{
   while (true) {
      <think>;
      get forks(k);          /* client requests two forks via monitor */
      <eat spaghetti>;
      release forks(k);      /* client releases forks via the monitor */
   }
}
```



Βασικά σημεία

- Η αναστολή είναι μόνιμη, εκτός και αν το λειτουργικό σύστημα αναλάβει έκτακτα μέτρα
- Το αδιέξοδο μπορεί να περιλαμβάνει επαναχρησιμοποιούμενους ή καταναλώσιμους πόρους
- Τρεις προσεγγίσεις υπάρχουν για την αντιμετώπιση των αδιεξόδων:
  - Η πρόληψη: το αδιέξοδο δεν θα συμβεί.
  - Η ανίχνευση: το ΛΣ είναι πάντα πρόθυμο να ικανοποιήσει τις απαιτήσεις σε πόρους.
  - Η αποφυγή: αναλύει κάθε νέα αίτηση, ώστε να αποφασίσει αν αυτή μπορεί να οδηγήσει σε αδιέξοδο.

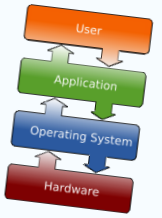


Αδιέξοδα – Άσκηση 1

Να σχεδιάσετε ένα γράφο εκχώρησης πόρων για τα παρακάτω:

- Η διεργασία P1 απαιτεί τον πόρο R1
- Η διεργασία P2 απαιτεί τον πόρο R3
- Ο πόρος R1 εκχωρείται στη διεργασία P2
- Ο πόρος R2 εκχωρείται στη διεργασία P1
- Ο πόρος R3 εκχωρείται στη διεργασία P3

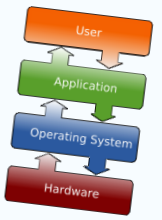
Υπάρχει αδιέξοδο;



Αδιέξοδα – Άσκηση 2

Θεωρείστε ένα σύστημα με 4 διεργασίες (A, B, C, D) και 5 επαναχρησιμοποιήσιμους πόρους (R0, R1, R2, R3, R4). Η τρέχουσα ανάθεση πόρων (τρέχουσα κατάσταση) στις διεργασίες δίνεται από τον παρακάτω πίνακα :

Εκχώρηση	R0	R1	R2	R3	R4
A	1	0	2	1	1
B	2	0	1	1	0
C	1	1	0	1	0
D	1	1	1	1	0

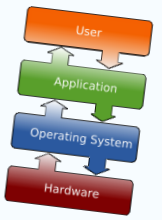


Αδιέξοδα – Άσκηση 2 (συν.)

Θεωρείστε επίσης ότι οι μέγιστες απαιτήσεις των διεργασιών δίνονται από τον παρακάτω πίνακα:

Μέγιστη απαίτηση	R0	R1	R2	R3	R4
A	1	1	2	1	2
B	2	2	2	1	0
C	2	1	3	1	0
D	1	1	2	2	1

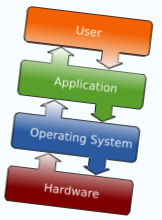
Οι διαθέσιμοι προς εκχώρηση πόροι είναι : **0 0 x 1 1**
Να βρεθεί η μικρότερη τιμή **x** για την οποία το σύστημα θα οδηγηθεί σε ασφαλή κατάσταση.



Αδιέξοδα – Άσκηση 3

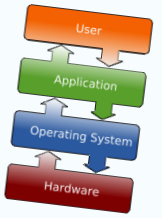
- Θεωρείστε τις διεργασίες P και Q και τους σημαφόρους s και t με αρχικές τιμές ίσες με 1. Οι δύο διεργασίες εκτελούν τα βήματα που φαίνονται παρακάτω. Να βρείτε μια ακολουθία εκτέλεσης των βημάτων των δύο διεργασιών που οδηγεί σε αδιέξοδο. Να γράψετε τις διαδοχικές τιμές που λαμβάνουν οι σημαφόροι.

P	Q
Step P1: WAIT(s)	Step Q1: WAIT(s)
Step P2: WAIT(t)	Step Q2: WAIT(t)
Step P3: Critical section PA	Step Q3: Critical section QA
Step P4: SIGNAL(s)	Step Q4: SIGNAL(t)
Step P5: WAIT(s)	Step Q5: SIGNAL(s)
Step P6: Critical section PB	Step P7: SIGNAL(s)
Step P8: SIGNAL(t)	



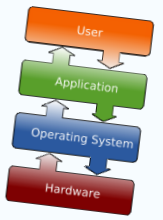
Σύγχρονοι μηχανισμοί ταυτοχρονισμού

- Σε συστήματα με UNIX ΛΣ
- Σε συστήματα με Linux ΛΣ
- Σε συστήματα με Windows ΛΣ



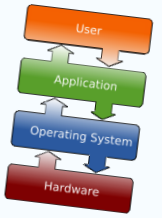
Οι μηχανισμοί ταυτοχρονισμού σε UNIX

- Δίαυλοι (Pipes)
- Μηνύματα
- Διαμοιραζόμενη μνήμη
- Σηματοφόροι
- Σήματα



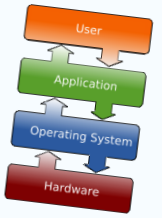
Σήματα στο UNIX

Value	Name	Description
01	SIGHUP	Hang up; sent to process when kernel assumes that the user of that process is doing no useful work
02	SIGINT	Interrupt
03	SIGQUIT	Quit; sent by user to induce halting of process and production of core dump
04	SIGILL	Illegal instruction
05	SIGTRAP	Trace trap; triggers the execution of code for process tracing
06	SIGIOT	IOT instruction
07	SIGEMT	EMT instruction
08	SIGFPE	Floating-point exception
09	SIGKILL	Kill; terminate process
10	SIGBUS	Bus error
11	SIGSEGV	Segmentation violation; process attempts to access location outside its virtual address space
12	SIGSYS	Bad argument to system call
13	SIGPIPE	Write on a pipe that has no readers attached to it
14	SIGALRM	Alarm clock; issued when a process wishes to receive a signal after a period of time
15	SIGTERM	Software termination
16	SIGUSR1	User-defined signal 1
17	SIGUSR2	User-defined signal 2
18	SIGCHLD	Death of a child
19	SIGPWR	Power failure



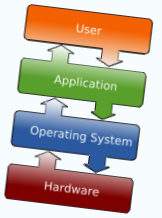
Οι μηχανισμοί ταυτοχρονισμού σε LINUX

- Περιλαμβάνει όλους τους μηχανισμούς που υπάρχουν στο UNIX
- Οι ατομικές λειτουργίες εκτελούνται χωρίς διακοπές και χωρίς να επεμβαίνουν σε άλλες



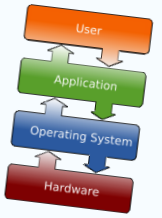
Ατομικές λειτουργίες σε Linux

Atomic Integer Operations	
`ATOMIC_INIT (int i)`	At declaration: initialize an atomic_t to i
`int atomic_read(atomic_t *v)`	Read integer value of v
`void atomic_set(atomic_t *v, int i)`	Set the value of v to integer i
`void atomic_add(int i, atomic_t *v)`	Add i to v
`void atomic_sub(int i, atomic_t *v)`	Subtract i from v
`void atomic_inc(atomic_t *v)`	Add 1 to v
`void atomic_dec(atomic_t *v)`	Subtract 1 from v
`int atomic_sub_and_test(int i, atomic_t *v)`	Subtract i from v; return 1 if the result is zero; return 0 otherwise
`int atomic_add_negative(int i, atomic_t *v)`	Add i to v; return 1 if the result is negative; return 0 otherwise (used for implementing semaphores)
`int atomic_dec_and_test(atomic_t *v)`	Subtract 1 from v; return 1 if the result is zero; return 0 otherwise
`int atomic_inc_and_test(atomic_t *v)`	Add 1 to v; return 1 if the result is zero; return 0 otherwise



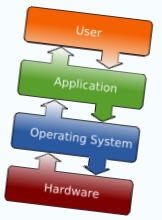
Ατομικές λειτουργίες σε Linux

Atomic Bitmap Operations	
`void set_bit(int nr, void *addr)`	Set bit nr in the bitmap pointed to by addr
`void clear_bit(int nr, void *addr)`	Clear bit nr in the bitmap pointed to by addr
`void change_bit(int nr, void *addr)`	Invert bit nr in the bitmap pointed to by addr
`int test_and_set_bit(int nr, void *addr)`	Set bit nr in the bitmap pointed to by addr; return the old bit value
`int test_and_clear_bit(int nr, void *addr)`	Clear bit nr in the bitmap pointed to by addr; return the old bit value
`int test_and_change_bit(int nr, void *addr)`	Invert bit nr in the bitmap pointed to by addr; return the old bit value
`int test_bit(int nr, void *addr)`	Return the value of bit nr in the bitmap pointed to by addr



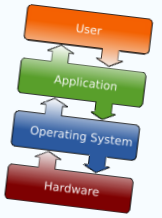
Spinlocks σε Linux

`void spin_lock(spinlock_t *lock)`	Acquires the specified lock, spinning if needed until it is available
`void spin_lock_irq(spinlock_t *lock)`	Like spin_lock, but also disables interrupts on the local processor
`void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)`	Like spin_lock_irq, but also saves the current interrupt state in flags
`void spin_lock_bh(spinlock_t *lock)`	Like spin_lock, but also disables the execution of all bottom halves
`void spin_unlock(spinlock_t *lock)`	Releases given lock
`void spin_unlock_irq(spinlock_t *lock)`	Releases given lock and enables local interrupts
`void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)`	Releases given lock and restores local interrupts to given previous state
`void spin_unlock_bh(spinlock_t *lock)`	Releases given lock and enables bottom halves
`void spin_lock_init(spinlock_t *lock)`	Initializes given spinlock
`int spin_trylock(spinlock_t *lock)`	Tries to acquire specified lock; returns nonzero if lock is currently held and zero otherwise
`int spin_is_locked(spinlock_t *lock)`	Returns nonzero if lock is currently held and zero otherwise



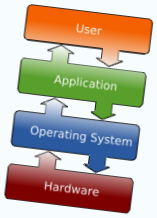
Σημαφόροι σε Linux Semaphores

Traditional Semaphores	
`void sema_init(struct semaphore *sem, int count)`	Initializes the dynamically created semaphore to the given count
`void init_MUTEX(struct semaphore *sem)`	Initializes the dynamically created semaphore with a count of 1 (initially unlocked)
`void init_MUTEX_LOCKED(struct semaphore *sem)`	Initializes the dynamically created semaphore with a count of 0 (initially locked)
`void down(struct semaphore *sem)`	Attempts to acquire the given semaphore, entering uninterruptible sleep if semaphore is unavailable
`int down_interruptible(struct semaphore *sem)`	Attempts to acquire the given semaphore, entering interruptible sleep if semaphore is unavailable; returns -EINTR value if a signal other than the result of an up operation is received.
`int down_trylock(struct semaphore *sem)`	Attempts to acquire the given semaphore, and returns a nonzero value if semaphore is unavailable
`void up(struct semaphore *sem)`	Releases the given semaphore
Reader-Writer Semaphores	
`void init_rwsem(struct rw_semaphore, *rwsem)`	Initalizes the dynamically created semaphore with a count of 1
`void down_read(struct rw_semaphore, *rwsem)`	Down operation for readers
`void up_read(struct rw_semaphore, *rwsem)`	Up operation for readers
`void down_write(struct rw_semaphore, *rwsem)`	Down operation for writers
`void up_write(struct rw_semaphore, *rwsem)`	Up operation for writers



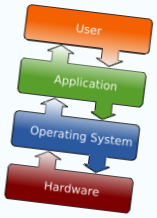
Λειτουργίες εμποδισμού μνήμης (Memory Barrier) σε Linux

`rmb()`	Prevents loads from being reordered across the barrier
`wmb()`	Prevents stores from being reordered across the barrier
`mb()`	Prevents loads and stores from being reordered across the barrier
`Barrier()`	Prevents the compiler from reordering loads or stores across the barrier
`smp_rmb()`	On SMP, provides a `rmb()` and on UP provides a `barrier()`
`smp_wmb()`	On SMP, provides a `wmb()` and on UP provides a `barrier()`
`smp_mb()`	On SMP, provides a `mb()` and on UP provides a `barrier()`

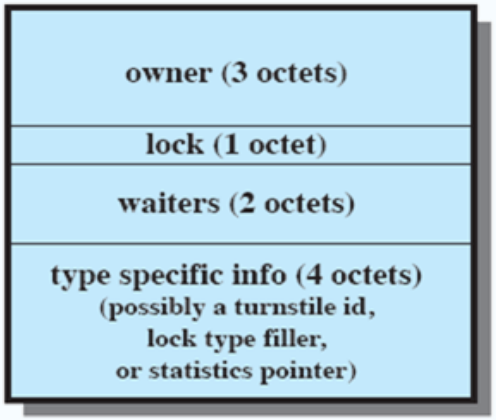


Αρχές συγχρονισμού σε Solaris

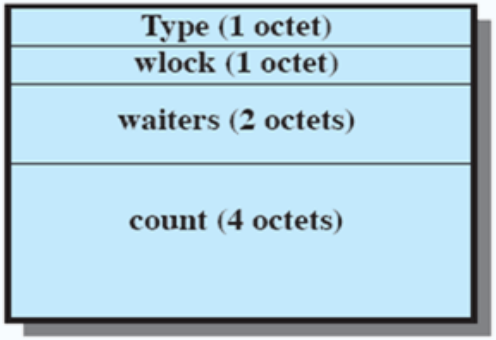
- Mutual exclusion (mutex) locks
- Semaphores
- Multiple readers, single writer (readers/writer) locks
- Condition variables



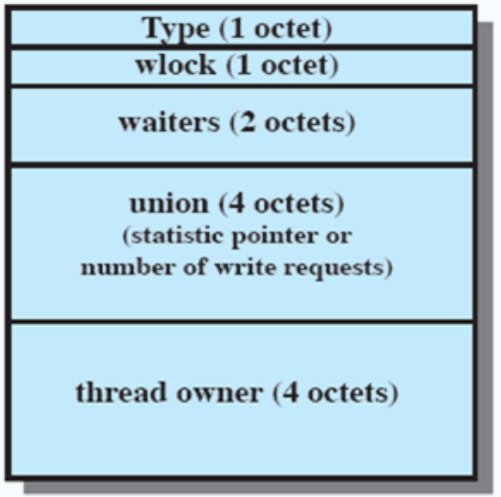
Δομές δεδομένων συγχρονισμού σε Solaris



(a) MUTEX lock

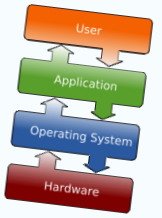


(b) Semaphore



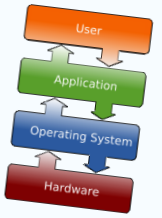
(c) Reader/writer lock

(d) Condition variable



Αντικείμενα συγχρονισμού σε Windows

Object Type	Definition	Set to Signaled State When	Effect on Waiting Threads
Notification Event	An announcement that a system event has occurred	Thread sets the event	All released
Synchronization event	An announcement that a system event has occurred.	Thread sets the event	One thread released
Mutex	A mechanism that provides mutual exclusion capabilities; equivalent to a binary semaphore	Owning thread or other thread releases the mutex	One thread released
Semaphore	A counter that regulates the number of threads that can use a resource	Semaphore count drops to zero	All released
Waitable timer	A counter that records the passage of time	Set time arrives or time interval expires	All released
File	An instance of an opened file or I/O device	I/O operation completes	All released
Process	A program invocation, including the address space and resources required to run the program	Last thread terminates	All released
Thread	An executable entity within a process	Thread terminates	All released



Αναφορές

- “Λειτουργικά Συστήματα – Αρχές Σχεδίασης”, 4η έκδοση, W. Stallings, Εκδόσεις Τζιόλα, 2008.
- “Operating System Concepts”, 7η έκδοση, από Abraham Silberschatz, Peter Galvin και Greg Gagne, Addison-Wesley, 2004.
- “Operating Systems: Design and Implementation”, 3η έκδοση, από Andrew Tanenbaum και Albert Woodhull, Prentice Hall, 2006.
- Διαφάνειες Δ. Κεχαγιάς, “Λειτουργικά Συστήματα”, 2007.