Προηγμένες αρχιτεκτονικές υπολογιστών και προγραμματισμός παραλλήλων συστημάτων

09. Παράλληλος Προγραμματισμός

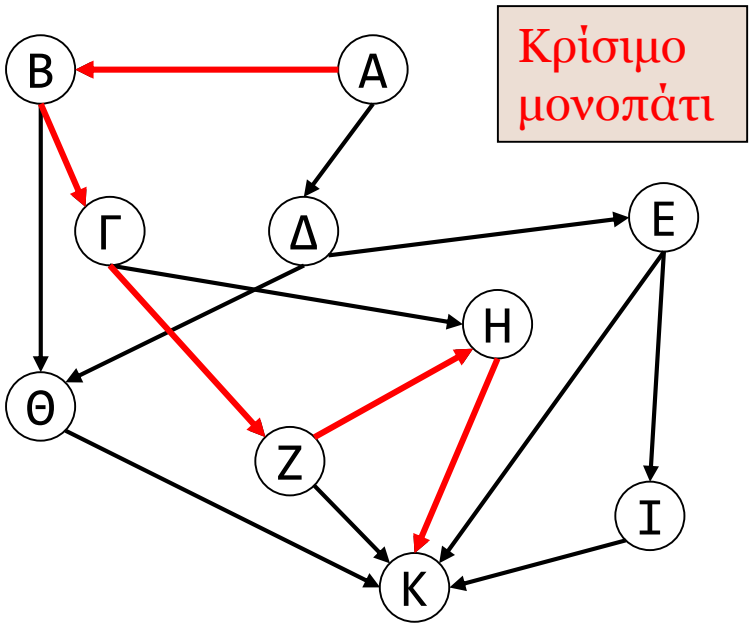
Ο νόμος του Amdahl - υπενθύμιση

- Κάθε αλγόριθμος έχει ένα ποσοστό s που δεν παραλληλίζεται (σειριακό τμήμα του αλγορίθμου).
- Ακόμα και αν αγνοήσουμε το χρόνο επικοινωνίας και αν υποθέσουμε ότι τεμαχίσαμε τον κώδικα σε απολύτως ίσα κομμάτια η επιτάχυνση που θα επιτύχουμε δεν είναι μεγαλύτερη από $1/s$.

$$\text{Επιτάχυνση} < 1/s$$

- Πχ. $s = 0.05$ (δηλαδή $s = 5\%$) τότε Επιτάχυνση < 20, όσους επεξεργαστές N και αν χρησιμοποιήσουμε για την παράλληλη εκτέλεση.

Ο νόμος του Amdahl – Παράδειγμα



Γράφος εξάρτησης

Έστω ότι κάθε κόμβος απαιτεί χρόνο t=1. Σειριακή εκτέλεση T=10.

Παράλληλη εκτέλεση όχι μικρότερη από το κρίσιμο μονοπάτι Α-Β-Γ-Ζ-Η-Κ, μήκος = 6, $T_{παρ} \geq 6$

$s = 6/10$

Επιτάχυνση $\leq 1/s = 10/6$

Εργασίες, Διεργασίες

- **Εργασία (task):** το μικρότερο κομμάτι κώδικα που δε διασπάται σε μικρότερα τμήματα = κόμβος στο Γράφο Εξάρτησης.
- **Διεργασία (process):** σύνολο από εργασίες που εκτελούνται στον ίδιο επεξεργαστή.

Επικοινωνία διεργασιών

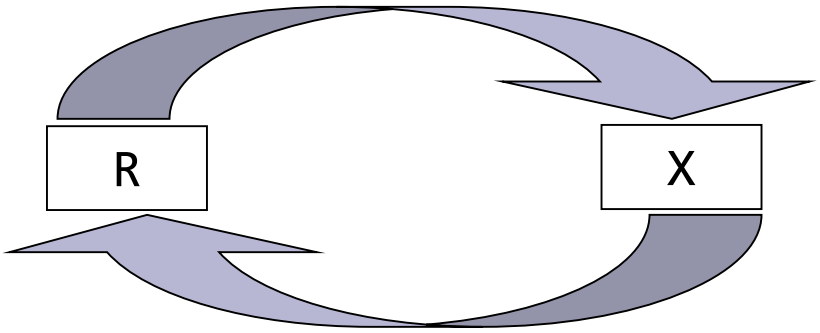
- Γιατί επικοινωνία;
- Χρήση κοινών αγαθών (κοινές μεταβλητές, κοινές θύρες I/O, κοινοί πόροι, κλπ) όταν έχουμε κοινή μνήμη.
- Ανταλλαγή μηνυμάτων όταν έχουμε κατανεμημένη μνήμη.
- Συγχρονισμός.

Υποστήριξη από hardware

- Βασικές λειτουργίες
  - Ατομική ανταλλαγή (atomic exchange)
  - Διάβασε και αύξησε (fetch-and-increment)
  - Έλεγξε και γράψε (test-and-set)

Atomic Exchange

- Καταχωρητής $R \leftrightarrow$ Θέση μνήμης X
- αμοιβαία ανταλλαγή δεδομένων σε ένα **αδιάσπαστο** (*ατομικό*) κύκλο.

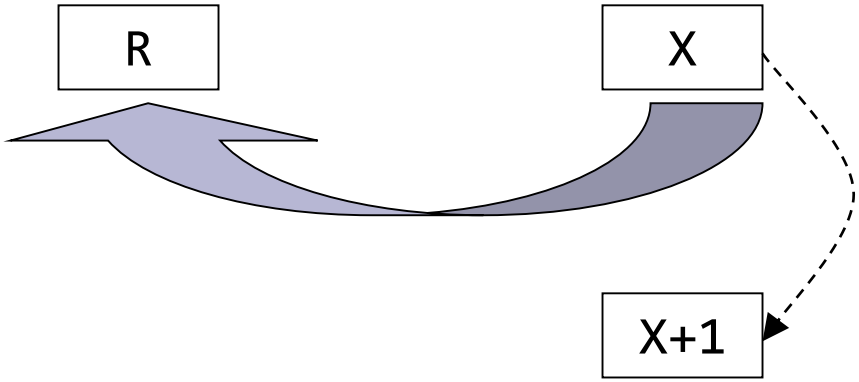


Fetch and Increment

- Καταχωρητής *R*, Θέση μνήμης *X*.

```
R = X
X = X+1
```

αδιάσπαστη εκτέλεση για την αποφυγή συνθηκών data race (race conditions).

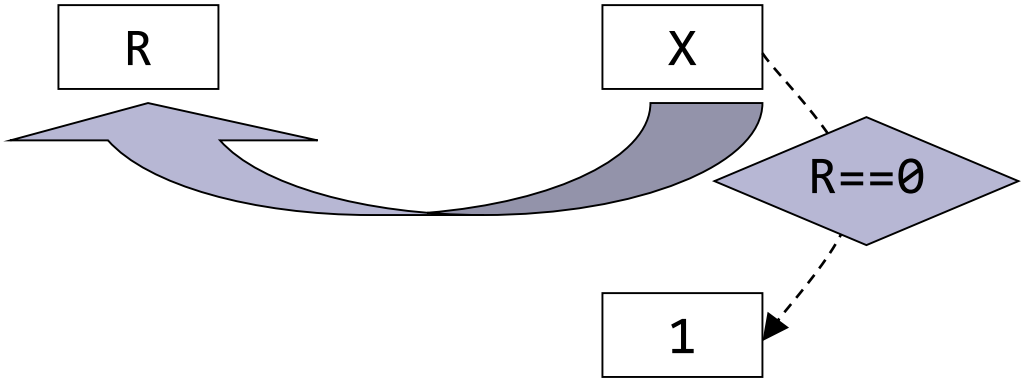


Test and Set

- Καταχωρητής R, Θέση μνήμης X.

```
R = X
if (R==0) X = 1
```

αδιάσπαστη εκτέλεση για την αποφυγή συνθηκών data race (race conditions).



Χρήση κοινών αγαθών

- Πρόβλημα στη χρήση κοινών αγαθών (data race).
- Έστω C = κάποιος μετρητής.

Επεξεργαστής A	**Επεξεργαστής B**
`load R5,C`	`load R2,C`
`add R5,1`	`add R2,1`
`store C,R5`	`store C,R2`

- Ανάλογα με τη σειρά εκτέλεσης, `C = C + 1`, ή `C = C + 2`.

Κλείδωμα - Ξεκλείδωμα

- Προστασία κοινών μεταβλητών.
- Σηματοφορέας (Semaphore) S:
  - `S = 1` : κλειδωμένο, απαγορεύεται η πρόσβαση από άλλον
  - `S = 0` : ξεκλείδωτο, επιτρέπεται η πρόσβαση

- **Κλείδωμα:** `lock(S)` (γνωστό και ως P(S)).
  - Λειτουργία:

```
while (S==1) { wait; }
S=1;
```

- **Ξεκλείδωμα:** `unlock(S)` (γνωστό και ως V(S))
  - Λειτουργία:

```
S=0;
```

Πρακτικό κλείδωμα (spin lock)

- Χρήση atomic exchange (`aexch`):

```
        mov R1,#1  ; R1 = 1: τιμή κλειδώματος.
wait:   aexch R1,S ; ατομική ανταλλαγή μεταξύ R1 και S.
        bneqz R1,wait ; αν S≠0 τότε έχει κλειδωθεί από
                      ; άλλη διεργασία ξαναπροσπάθησε
                      ; (goto wait).
        ……..          ; υπόλοιπος κώδικας
```

Πρακτικό κλείδωμα (spin lock)

- Δεν απαιτείται atomic exchange (aexch) αλλά δε βλάπτει:

```
mov R1,#0  ; R1 = 0: τιμή ξεκλειδώματος.
aexch R1,S ; ατομική ανταλλαγή μεταξύ R1 και S.
.....      ; υπόλοιπος κώδικας
```

load-linked και store-conditional

- **load-linked:** ίδια με την απλή load αλλά ενημερώνεται ο link-register με την τιμή της διεύθυνσης μνήμης από όπου γίνεται το load.
- **link-register:** παρακολουθεί συνεχώς τη δραστηριότητα του bus και αν δει να κυκλοφορεί διεύθυνση ίση με το περιεχόμενό του τότε αυτομάτως μηδενίζεται.
- **store-conditional:** ίδια με τη store αλλά αν link-register == 0 τότε το store αποτυγχάνει (δεν εκτελείται) και επιστρέφεται κάποια χαρακτηριστική τιμή (πχ. NaN, Inf, κλπ).

Κλείδωμα με ll - sc

- ll: load-linked.
- sc: store-conditional.

```
     ld R4,#1     ; R4 = 1.
try: mov R3,R4    ; φόρτωσε προσωρινά τον R4 στον R3.
     ll R2,S      ; load-linked.
     sc R3,S      ; store-conditional.
     beqz R3,try; άλμα πίσω αν store απέτυχε (R3=0).
     mov R4,R2    ; μεταφορά του παλαιού περιεχομένου
                  ; του S στον R4.
```

Φράγμα συγχρονισμού

- Το φράγμα συγχρονισμού *N* διεργασιών είναι ένα σημείο στον κώδικα όπου το πρόγραμμα σταματάει μέχρι να φτάσουν στο ίδιο φράγμα οι υπόλοιπες *N-1* διεργασίες.
- Συναντάται σε loops και σε άλλα σημεία που απαιτείται συγχρονισμός μεταξύ διεργασιών.

Απλό (εσφαλμένο) φράγμα συγχρονισμού

```
lock(S_count);                  /* προστάτεψε τη μεταβλητή count. */
if (count==0) release=0;        /* Αρχικοποίηση του release. */
                                /* release=0 -> σταμάτα και περίμενε. */
                                /* release=1 -> OK, προχώρα. */
count = count +1;               /* μέτρα πόσοι έφτασαν στο φράγμα */
unlock(S_count);                /* τέλος κρίσιμου τμήματος για το count */
if (count==total) {             /* έφτασαν όλοι στο φράγμα */
  count=0;                      /* μηδένισε το μετρητή */
  release=1;                    /* απελευθέρωσε τις διεργασίες */
}
else spin(release);             /* περίμενε μέχρι release=1 */
```

Πρόβλημα απλού φράγματος συγχρονισμού

- Χρήση φράγματος συγχρονισμού συνήθως σε loop.
- Το πρόβλημα εμφανίζεται σε περίπτωση που το ίδιο φράγμα χρησιμοποιείται σε δύο διαδοχικά iterations.

Παράδειγμα:

- Δύο διεργασίες: μία αργή και μία γρήγορη:
  - Η αργή φτάνει πρώτη -> release = 0.
  - Η γρήγορη φτάνει δεύτερη, θέτει release = 1, και εκτελεί κάποιο άλλο τμήμα κώδικα.
  - Επανέρχεται στο φράγμα στο επόμενο iteration, θέτει release = 0 και μπαίνει σε αναμονή.
  - Η αργή τώρα διαβάζει το release και το βλέπει πάλι 0.
  - Και οι 2 διεργασίες τώρα είναι σε αδιέξοδο (deadlock) και δε θα βγουν ποτέ.

Ορθό φράγμα συγχρονισμού

- Εναλλαγή σημασίας release:
  - iteration *k*: release=0: stop, release=1: go
  - iteration *k*+1: release=1:stop, release=0:go

release

A	0
B	0
Γ	0
Δ	1

iteration *k*
local_sense=1

release

A	1
B	1
Γ	1
Δ	0

iteration *k*+1
local_sense=0

. . . .
.

Πρακτικό (ορθό) φράγμα συγχρονισμού

```
local_sense=1-local_sense;    /*local_sense = τοπική μεταβλητή */
lock(S_count);                /* προστάτεψε την μεταβλητή count */
if (count==0) {               /* αρχικοποίηση του release */
  release=1 - local_sense;    /* release = local_sense -> προχώρα */
}                             /* release = 1-local_sense -> περίμενε */
count = count + 1;            /* μέτρα πόσοι έφτασαν στο φράγμα */
unlock(S_count);              /* τέλος κρίσιμου τμήματος για το count */
if (count==total) {           /* έφτασαν όλοι στο φράγμα */
  count=0;                    /* μηδένισε το μετρητή */
  release=local_sense;        /* απελευθέρωσε τις διεργασίες */
}
else spin(release==local_sense);    /* περίμενε */
```

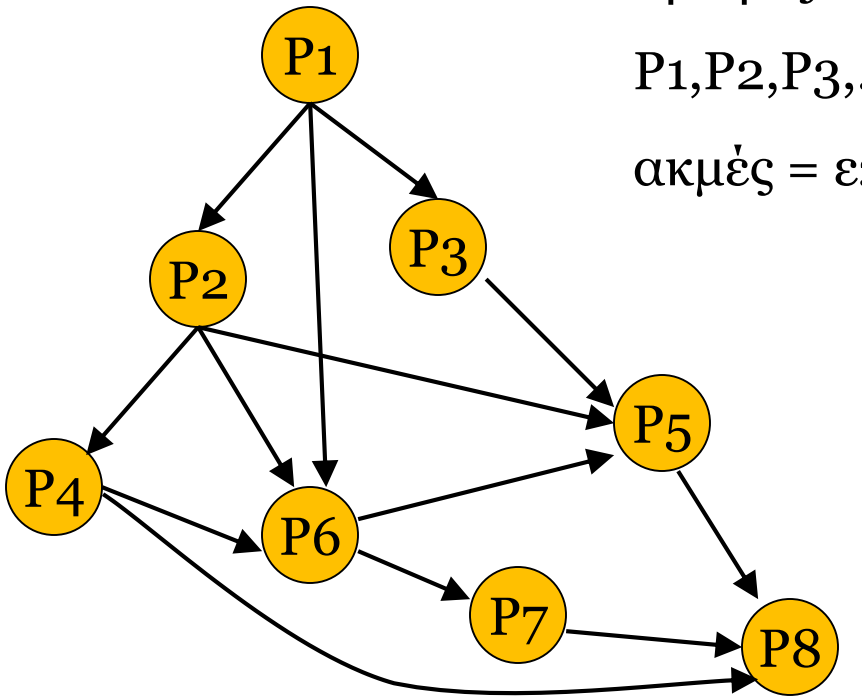
Παράλληλες εντολές υψηλού επιπέδου

- Το ζεύγος εντολών `fork` – `join`.
- Το ζεύγος εντολών `parbegin` – `parend`.
- Η δομή monitor.

Οι εντολές `fork` και `join`

- `Fork label`: δημιούργησε αντίγραφο του τρέχοντος προγράμματος και άρχισε να το εκτελείς από το σημείο με ετικέτα `label`.
- `join n`: φράγμα συγχρονισμού για n διεργασίες.

Παράδειγμα fork - join (1)

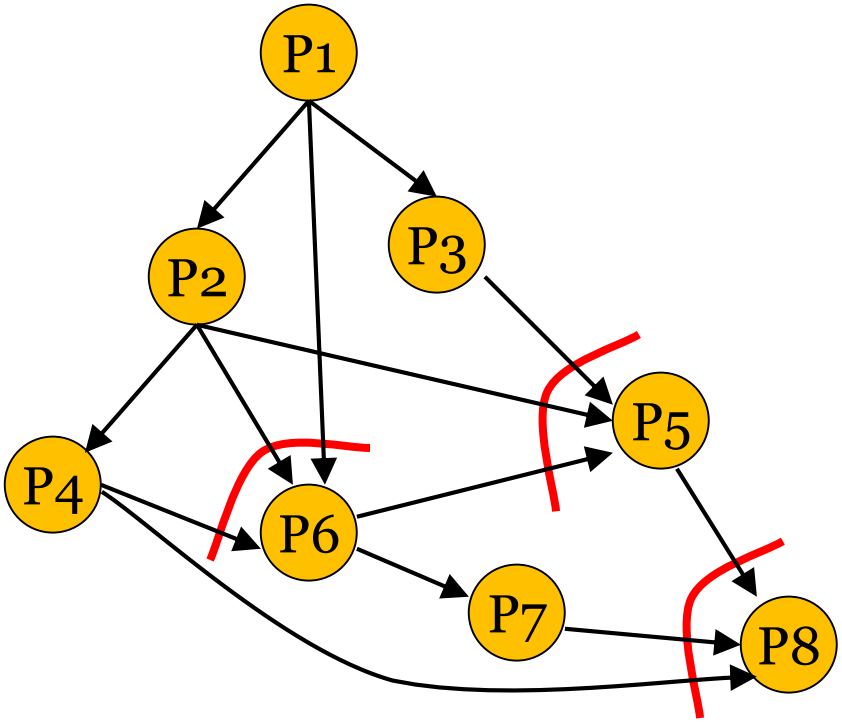


Γράφος εξάρτησης:

P1,P2,P3,...= εργασίες = ρουτίνες

ακμές = επικοινωνία δεδομένων

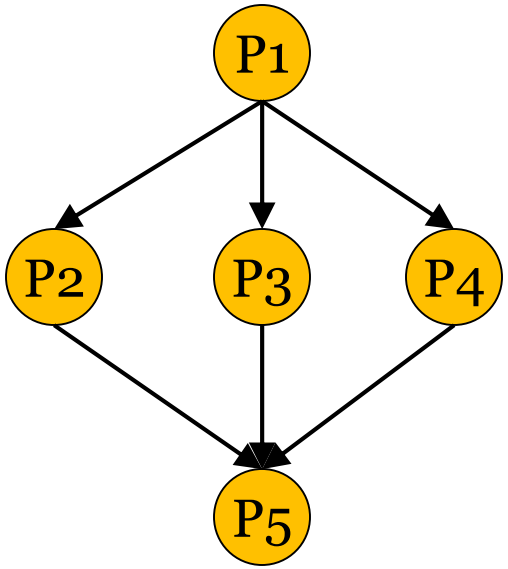
Παράδειγμα fork - join (2)



Οι εντολές parbegin και parend

- parbegin - parend: όπως τα begin και end σε ένα loop μόνο που οι διεργασίες δεν εκτελούνται σειριακά αλλά παράλληλα.
- parend: φράγμα συγχρονισμού για τις διεργασίες.

Παράδειγμα parbegin – parend




```
P1;
parbegin
P2; || P3; || P4;
parend
P5;
```

Αδιέξοδα (deadlocks)

- Αδιέξοδο: κατάσταση κατά την οποία μια σειρά εργασιών περιμένουν με κυκλικό τρόπο η μια την άλλη και καμία δεν μπορεί να προχωρήσει.
- Τέσσερεις συνθήκες που οδηγούν σε αδιέξοδο:
  - Δέσμευση και αναμονή.
  - Όχι προαπελευθέρωση.
  - Αμοιβαίος αποκλεισμός.
  - Κυκλική αναμονή.

Αποφυγή αδιεξόδων

- Στατική αποφυγή:
  - Οργάνωση του κώδικα έτσι ώστε σε καμία περίπτωση να μην υπάρχει πιθανότητα αδιεξόδου.
  - Λύση compile-time.
- Δυναμική αποφυγή:
  - Αφήνουμε τον κώδικα όπως είναι αλλά έχουμε μηχανισμούς ανίχνευσης αδιεξόδων και επίλυσής τους αν προκύψουν.
  - Λύση run-time.

Στατική αποφυγή αδιεξόδων

- Βασική υπόθεση: οι κώδικες των διεργασιών εκτελούνται ασύγχρονα.
- Λύση compile-time: αναδιοργάνωση των κρίσιμων τμημάτων έτσι ώστε να μην υπάρξει περίπτωση κυκλικής αναμονής: Μετακίνηση των lock νωρίτερα, μετακίνηση των unlock αργότερα.
- Χρήση των γράφων δέσμευσης αγαθών.

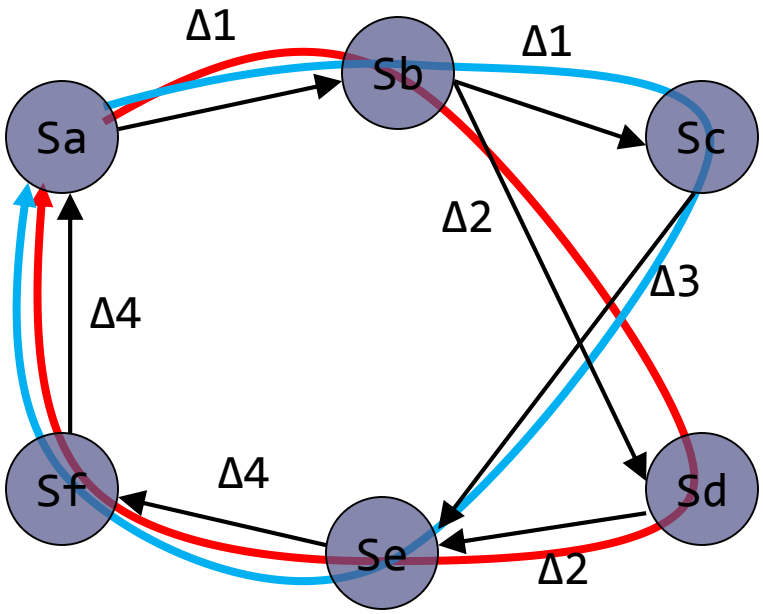
Γράφος δέσμευσης αγαθών (RAG)

- Resource Allocation Graph.
- Κόμβοι: κοινά αγαθά, πχ. σηματοφορείς S1, S2, S3, κλπ.
- Ακμές: μεταξύ S1, S2 αν υπάρχουν διαδοχικά κλειδώματα `P(S1)`, `P(S2)` χωρίς να παρεμβάλλεται ξεκλείδωμα `V(S1)`.
- Υπάρχει κίνδυνος αδιεξόδου αν και μόνο αν υπάρχει κύκλος στο γράφο δέσμευσης αγαθών.

Στατική αποφυγή αδιεξόδων (2)

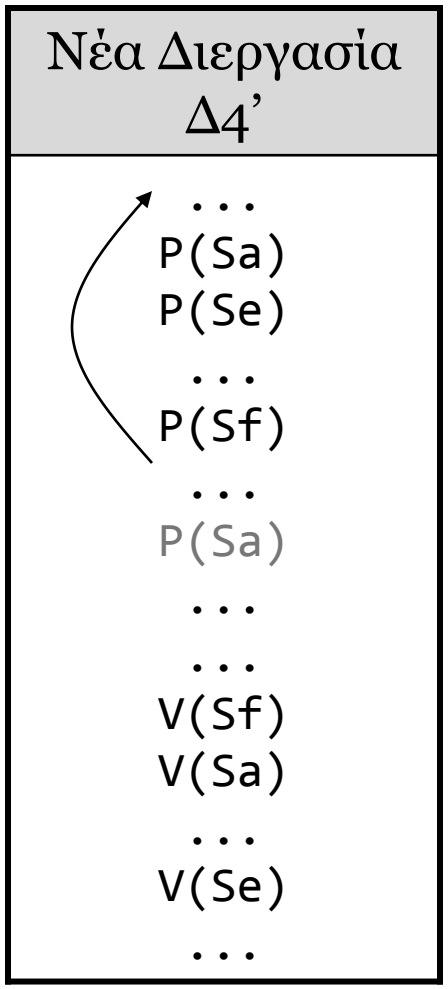
Διεργασία Δ1	Διεργασία Δ2	Διεργασία Δ3	Διεργασία Δ4
...	P(Sb)	...	...
P(Sa)	...	P(Sc)	...
...	P(Sd)	...	P(Se)
P(Sb)	...	P(Se)	...
...	P(Se)	...	P(Sf)
P(Sc)	...	...	...
...	...	...	P(Sa)
...	V(Se)	V(Se)	...
...	...	...	...
V(Sa)	...	...	V(Sf)
...	V(Sd)	...	V(Sa)
V(Sb)	...	V(Sc)	...
V(Sc)	V(Sb)	...	V(Se)
...	...	...	...

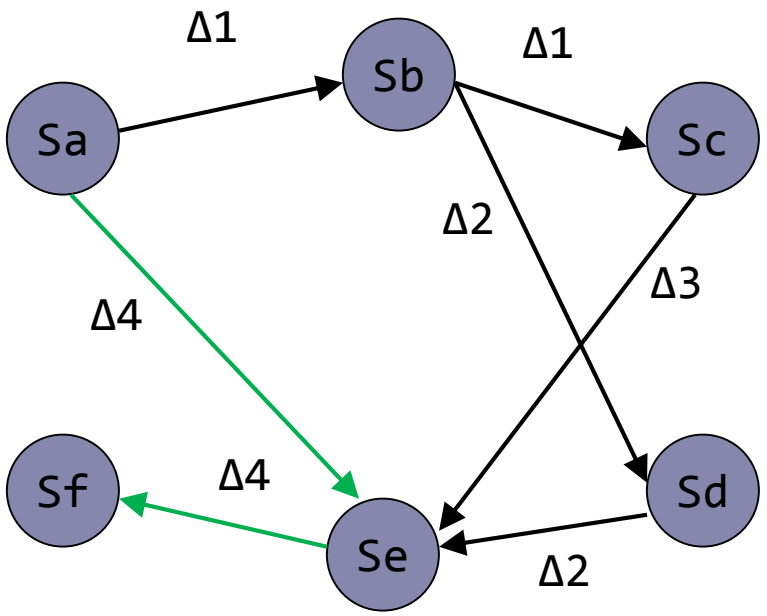
RAG - Παράδειγμα



Δύο κύκλοι = κίνδυνος αδιεξόδου

Στατική αποφυγή αδιεξόδων με RAG





Δυναμική αποφυγή αδιεξόδων

- Ανίχνευση αδιεξόδων:
  - Διαθέτουμε πίνακα αγαθών. Τα αγαθά χαρακτηρίζονται ως ελεύθερα ή δεσμευμένα.
  - Υπάρχουν αιτήσεις δέσμευσης αγαθών.
  - Ελέγχουμε την κατάσταση του πίνακα για κυκλική αναμονή.
- Επίλυση αδιεξόδου:
  - Όταν ανιχνευτεί αδιέξοδο επιλύεται σταματώντας κάποιες διεργασίες (ίσως όλες) και ελευθερώνοντας τα αγαθά τους
  - Επώδυνο. Αλγόριθμος βέλτιστης ανάκλησης διεργασιών.

Σύγκριση μεθόδων αποφυγής αδιεξόδων

- Δυναμική αποφυγή:
  - καλύτερη διαχείριση κοινών αγαθών.
  - δυσκολότερη υλοποίηση.
- Στατική αποφυγή:
  - απλούστερη υλοποίηση.
  - κλείδωμα κώδικα «χωρίς λόγο».