

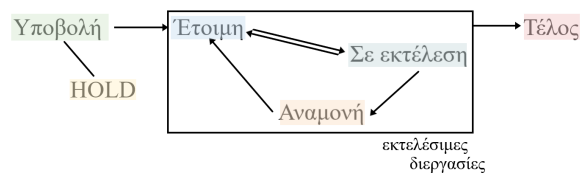
3 Διεργασίες

3 Χαρακτηριστικά διεργασιών

- ID
- Priority
- Context
- Λογιστικά στοιχεία

3 Καταστάσεις διεργασίας

- Υποβολή
- Τέλος
- HOLD
- Έτοιμη
- Σε εκτέλεση
- Σε αναμονή



4 Αλγόριθμος first come first served

4 Μετρικές

- turnaround time
 - waiting time
- σταθμισμένες μετρικές
- weighted turnaround time
 - weighted waiting time

6 Αλγόριθμος Round Robin

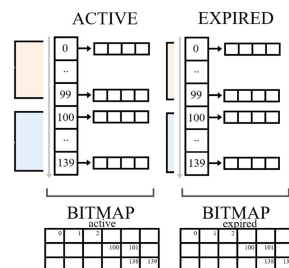
6 Ζητήματα απόδοσης

- βαθμός χρήσης (utilization)
- χρόνος απόκρισης (response time)

7 Χρονοδρομολογητής O(1)

7 Κατηγορίες διεργασιών

- 1) real time
- 2) normal time
 - a) CPU intensive
 - b) non-CPU intensive



10 Προτεραιότητες διεργασιών

- Αρχική ή σταθερή προτεραιότητα
- Δυναμική προτεραιότητα

$$DP = \text{MAX}[100, \{\min (SP - \text{bonus} + 5, 139)\}]$$

Αν $PR < 120$ τότε έχουμε $(140 - PR) * 20\text{ms}$

Αν $PR > 120$ τότε έχουμε $(140 - PR) * 5\text{ms}$

11	Completely Fair Scheduler (CFS)
11	Virtual runtime
12	nice
12	weight
13	Target Latency
13	Minimum Granularity
14	ΚΡΙΣΙΜΟ ΤΜΗΜΑ
15	<u>Αυστηρή εν' αλλαγή</u>
17	<u>TSL (Test & Set Lock)</u>
18	<u>Peterson</u>
20	<u>ΣΗΜΑΤΟΦΟΡΕΙΣ</u>
	πρόβλημα παραγωγού - καταναλωτή
21	ΛΥΣΗ ΧΩΡΙΣ ΣΗΜΑΤΟΦΟΡΕΙΣ
24	ΛΥΣΗ ΜΕ ΣΗΜΑΤΟΦΟΡΕΙΣ
29	Κατανομή μνήμης
29	- Best-Fit
30	- First-Fit
30	- Worst-Fit
30	- Next-Fit
31	- μέθοδος των φίλων
32	Αντιμετώπιση κατακερματισμού - σελιδοποίηση
34	Transmission Lookaside Buffer (TLB)
35	ΑΡΧΕΙΑ
	άμεσοι δείκτες
	έμμεσοι δείκτες
	...
38	ΠΙΘΑΝΑ ΑΔΙΕΞΟΔΑ

Συντρέχουσες διεργασίες (concurrent ≠ parallel)

Παίρνουν εν' αλλάξ για λίγο χρόνο (κβάντο) τη CPU και τρέχουν.

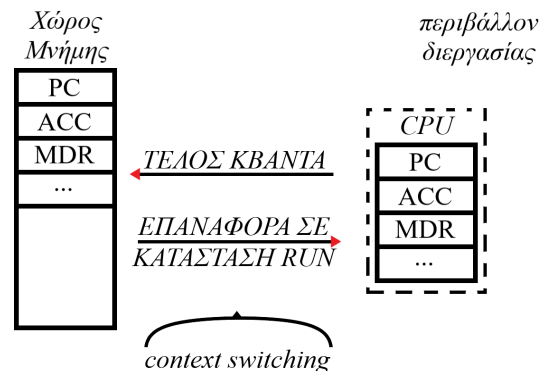
Διεργασίες

Αποτελούν προγράμματα που τρέχουν. (γενικός ορισμός)

Χαρακτηριστικά διεργασιών

- **ID** (αριθμός ταυτοποίησης)
- **Priority** (προτεραιότητα)
- **Context** (περιβάλλον διεργασίας)

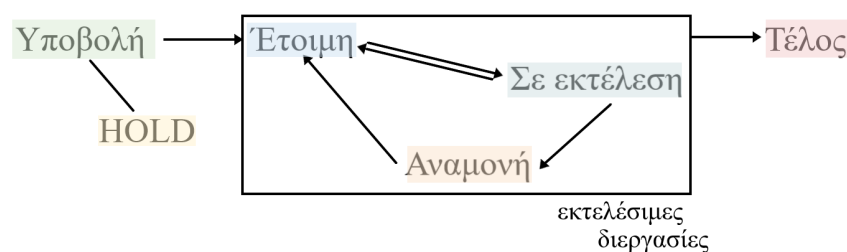
Κάθε διεργασία που τρέχει γράφει στους καταχωρητές της CPU (ACC, PC κτλ.). Η διεργασία κάποτε τελειώνει (κβάντα). Όταν η διεργασία τελειώσει, οι τιμές των καταχωρητών περνάνε στο χώρο μνήμης της διεργασίας.



Η εναλλαγή περιβάλλοντος “κοστίζει” στο χρήστη αφού, όταν γίνεται δεν τρέχει κανένα άλλο πρόγραμμα. Η εναλλαγή περιβάλλοντος εξαρτάται από τη ταχύτητα του επεξεργαστή.

- **Λογιστικά στοιχεία** (οι χρόνοι κάθε διεργασίας)

① Ο χρονοδρομολογητής (scheduler) δίνει χρόνο (κβάντα) στις διεργασίες. Ο χρονοδρομολογητής είναι πρόγραμμα του ΛΣ, και επομένως τρέχει “εις βάρος” του χρήστη. Γι’ αυτό πρέπει να είναι όσο πιο γρήγορος γίνεται.

Καταστάσεις διεργασίας

Υποβολή: Ο χρήστης ξεκινάει μία εφαρμογή

Τέλος: Ο χρήστης κλείνει την εφαρμογή

HOLD: Δημιουργείται χώρος μνήμης αλλά είμαστε σε κράτηση, όσο περιμένουμε το χρονοδρομολογητή (scheduler), ο οποίος τρέχει περιοδικά, να βάλει τη διεργασία στη λίστα εκτελέσιμων (οι διεργασίες που χρονοδρομολογούνται, δηλαδή παίρνουν κβάντα)

Έτοιμη: Διεργασία που είναι έτοιμη να τρέξει, έχει πάρει κβάντα και περιμένει

Σε εκτέλεση: Διεργασία που χρησιμοποιεί τη CPU μέχρι να τελειώσουν τα κβάντα

① Μετά την εκτέλεση υπάρχουν δύο περιπτώσεις:

- 1) Τέλος κβάντων -> Έτοιμη (γίνεται context switch)
- 2) **ΣΕ ANAMONH:** Η διεργασία περιμένει να συμβεί ένα γεγονός (π.χ. να γίνει accessible ένα μέρος μνήμης) για να γίνει ξανά Έτοιμη (γίνεται context switch)

Άσκηση

ID	Χρόνος Αφίξης	Χρόνος που απαιτείται για εκτέλεση
1	40	100
2	60	10
3	110	50

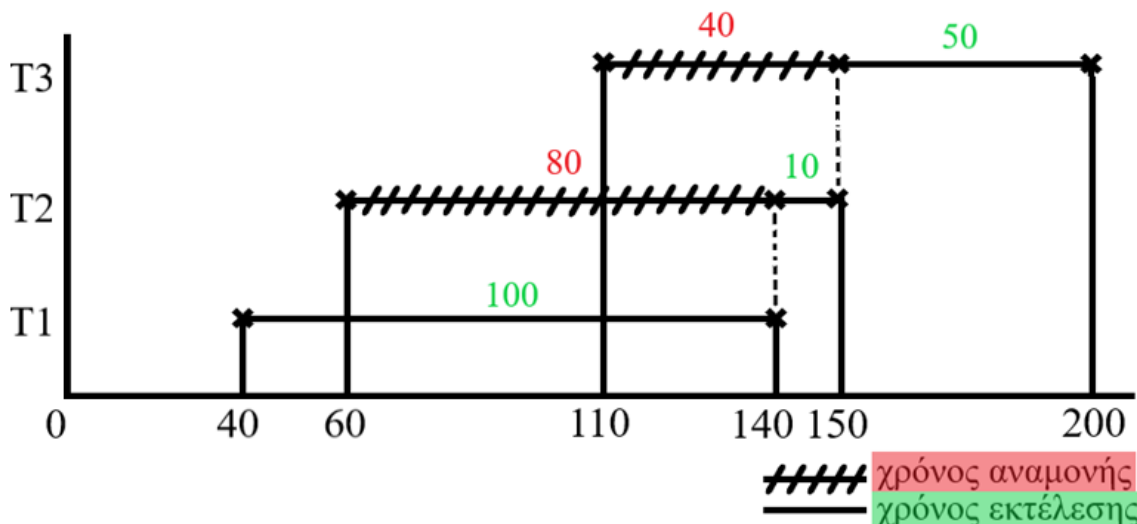
① Θεωρούμε πως ο **αλγόριθμος** ακολουθεί λογική **first come, first serve**, αν και αυτό γίνεται πολύ σπάνια στο πραγματικό κόσμο

Μετρικές

- **TT (turnaround time)**

real time

Χρόνος παραμονής: $TT = \text{χρόνος εξόδου} - \text{χρόνος εισόδου}$ ή $TT = WT + RT$
 συμβολικός χρόνος παραμονής μιας διεργασίας στο σύστημα



$$TT_1 = 140 - 40 = 100 \text{ ms}$$

$$TT_2 = 70 - 60 = 10 \text{ ms}$$

$$TT_3 = 160 - 110 = 50 \text{ ms}$$

- WT (waiting time)

Χρόνος αναμονής: $WT = TT - RT$

$$WT_1 = 100 - 100 = 0$$

$$WT_2 = 90 - 10 = 80$$

$$WT_3 = 90 - 50 = 40$$

① Υπάρχουν και τα ATT (average turnarround time) και AWT (average waiting time)

Σταθμισμένες (weighted) μετρικές

- WTT (weighted turnarround time)

Σταθμισμένος χρόνος παραμονής: $WTT = TT / RT$

π.χ. Αν $WTT = 10$, η διεργασία *βρίσκεται* στο σύστημα 10 φορές περισσότερο απ' όσο τρέχει

$$WTT_1 = 100 / 100 = 1$$

$$WTT_2 = 90 / 10 = 9$$

$$WTT_3 = 90 / 50 = 1,8$$

- WWT (weighted waiting time)

Σταθμισμένος χρόνος αναμονής: $WWT = WT / RT$

π.χ. Αν $WWT = 10$, η διεργασία *περιμένει* στο σύστημα 10 φορές περισσότερο απ' όσο τρέχει

$$WWT_1 = 0 / 100 = 0$$

$$WWT_2 = 80 / 10 = 8$$

$$WWT_3 = 40 / 50 = 0,8$$

① Υπάρχουν και τα AWTT (average WTT) και AWT (average WWT)

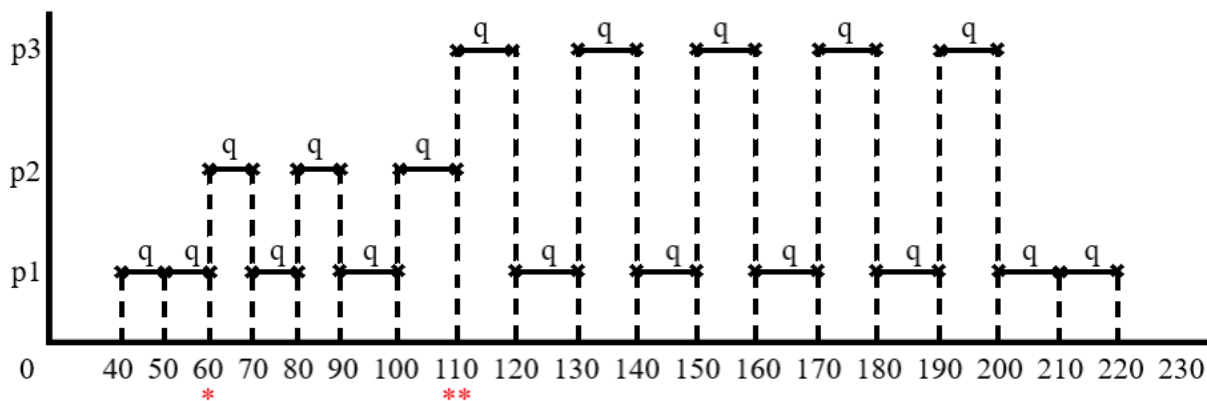
ID	Χρόνος Αφίξης	Χρόνος που απαιτείται για εκτέλεση
1	40	100
2	60	30
3	110	50

εκ περιτροπής
Αλγόριθμος Round Robin

quantum

① Υποθέτουμε πως έχει σταθερό κβάντο $q = 10$

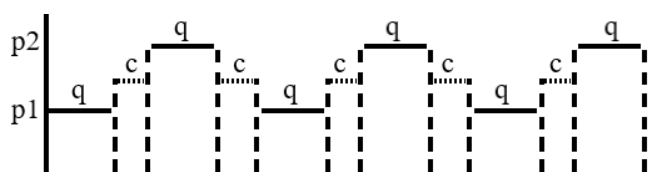
[στη πραγματικότητα το q κάθε διεργασίας διαφέρει, και εξαρτάται από πολλά πράγματα που θα δούμε παρακάτω]



* Στο $t = 60$ θα τρέξει πρώτα η $p2$ (αυτή που μπήκε τελευταία) και μετά από q θα τρέχουν εν' αλλάζ με την $p1$ ανά q

** Στο $t = 110$ θα τρέξει πρώτα η $p3$ (αυτή που μπήκε τελευταία) και μετά από q θα τρέχουν εν' αλλάζ με την $p1$ ανά q

Ζητήματα απόδοσης



c: context switch

• Βαθμός χρήσης (utilization)

$$U = q / (q+c)$$

πόσο ποσοστό του χρόνου CPU είναι χρήσιμο για κάθε διεργασία

π.χ. έστω $c = 5$ και $q1 = 100$ και $q2 = 10$

επομένως $U_1 = 100 / (100 + 5) = 95\%$ και $U_2 = 10 / (10 + 5) = 66\%$

άρα το U αυξάνει όσο αυξάνει το q .

- Χρόνος απόκρισης (response time)

ο χρόνος από τη στιγμή που ο χρήστης θα ζητήσει εκτέλεση μέχρι τη στιγμή που αποκρίνεται το πρόγραμμα

	$c = 5$	$q = 10$	$c = 5$	$q = 100$
p0	0	0		
p1	15	105		
p2	30	210		
p3	45	315		
...		
pn	n*15	n*105		

Όσο αυξάνεται το q, αυξάνεται ο χρόνος απόκρισης κάθε διεργασίας. Τελικά θέλω μεγάλο q για να αυξήσω το utilization και μικρό q για να μειώσω το response time των διεργασιών.

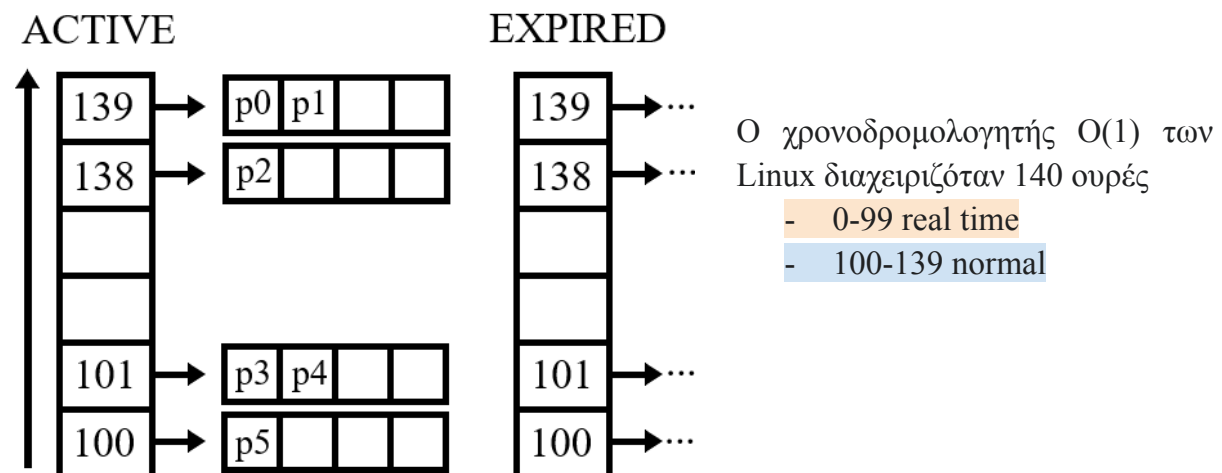
Χρονοδρομολογητής O(1)

Όλες οι διαδικασίες που απαιτούνται για την υλοποίηση της χρονοδρομολόγησης γίνονται σε σταθερό χρόνο (δηλ. δεν εξαρτάται από το πλήθος των διεργασιών για χρονοδρομολόγηση).

Κατηγορίες διεργασιών

- 1) **real time** (απόλυτη προτεραιότητα)
- 2) **normal** (οι διεργασίες του χρήστη)
 - a) CPU intensive
 - b) non-CPU intensive

① Διεργασίες που απασχολούν πολύ τη CPU έχουν κυρώσεις: χάνουν σε κβάντα. Δηλαδή το ΛΣ ευνοεί τις διεργασίες που είναι non-CPU intensive.



① Μεγαλύτερο priority έχουν τα μικρότερα νούμερα των ουρών. Επομένως η ουρά με το μεγαλύτερο priority είναι η 0 και αυτή με το μικρότερο η 139 (default priority val. = 120)

① Η προτεραιότητα των διεργασιών μπορεί να αλλάξει από το χρονοδρομολογητή.

- 1) Δημιουργούνται διεργασίες και μπαίνουν σε ουρές ανάλογα με τα priority τους.
- 2) Όταν εκτελείται ο χρονοδρομολογητής, εξετάζει ποια είναι η ουρά με το μικρότερο αριθμό που έχει διεργασίες προς εκτέλεση

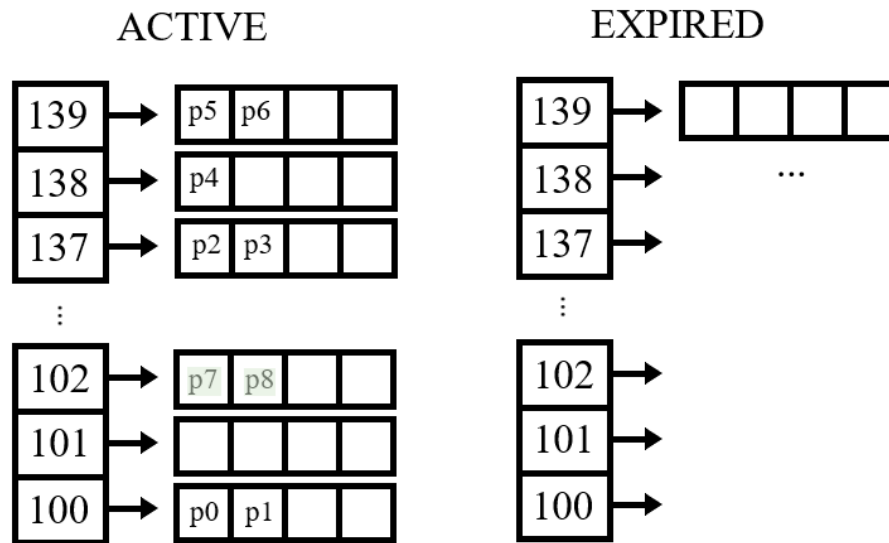
BITMAP (παράδειγμα)

0 ⁰	0 ¹	0 ²			
			1 ¹⁰⁰	0 ¹⁰¹	
				0 ¹³⁸	1 ¹³⁹

① Όταν μπει π.χ. στην ουρά 100 έστω μία διεργασία τότε BITMAP[100] ← 1

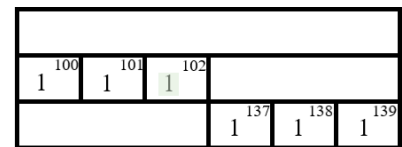
① Ο χρονοδρομολογητής ψάχνει το BITMAP από το [0] μέχρι να βρει τιμή 1. Είναι σταθερού χρόνου αφού δεν εξαρτάται από το πλήθος των διεργασιών.

Έχουμε 2 δομές με ουρές, την ACTIVE και την EXPIRED. Ο λόγος πίσω απ' αυτό, είναι θέμα ταχύτητας.



Βήματα

- 1) Έρχονται νέες διεργασίες και ενημερώνεται το BITMAP. Έστω p0-p6 έρχονται “ταυτόχρονα”.



- 2) Ο χρονοδρομολογητής διαβάζει το BITMAP και χρονοδρομολογεί βάση προτεραιοτήτων. Οι διεργασίες p7 και p8 δημιουργούνται και μέχρι να ξανατρέξει ο χρονοδρομολογητής, και να δει ότι είναι στην ουρά 102, βρίσκονται σε κατάσταση **HOLD** (δεν έχουν πάρει ακόμα κβάντα.)

① Όταν οι διεργασίες p7 και p8 δημιουργούνται, το BITMAP[102] γίνεται 1

Έστω σε χρόνο t τρέχει η p1 και όταν τα κβάντα της τελειώσουν, παίρνει τη CPU ο χρονοδρομολογητής. Εξετάζει τις ουρές και βλέπει 2 διεργασίες στην ουρά [102].

- 3) Οι p0 και p1 έχουν πάρει κβάντα και ολοκλήρωσαν (δεν έκλεισαν). Οι διεργασίες μεταφέρονται στις EXPIRED ουρές (όχι όμως απαραίτητα στην EXPIRED ουρά 100, αφού το σε ποια ουρά θα πάει εξαρτάται από το πόσο cpu-intensive είναι)

① Η δομή EXPIRED έχει διεργασίες που έχουν ολοκληρώσει τα κβάντα τους αλλά δεν έχουν κλείσει. Οι διεργασίες αυτές είναι σε κατάσταση ready (Ετοιμη).

① Όταν μια διεργασία ολοκληρώνει τα κβάντα της και μεταφέρεται στις EXPIRED, το BITMAP της ουράς στην οποία βρισκόταν (στη δομή ACTIVE) γίνεται 0 (εκτός αν υπάρχουν άλλες διεργασίες) και το BITMAP της EXPIRED ουράς που μεταφέρθηκε γίνεται 1.

- 4) Τελειώνουν τα κβάντα όλων των διεργασιών, και όλες βρίσκονται στη δομή EXPIRED. Το πρόβλημα είναι πως το να γίνουν ξανά ACTIVE οι διεργασίες που είναι EXPIRED και το πόσος χρόνος απαιτείται γι' αυτό, εξαρτάται από το πλήθος των διεργασιών, και επομένως δεν είναι σταθερού χρόνου.

Για να λύσουμε το πρόβλημα αυτό γίνεται εναλλαγή δεικτών μεταξύ της δομής ACTIVE και της δομής EXPIRED.

Προτεραιότητες διεργασιών

- Αρχική ή σταθερή προτεραιότητα (static priority SP)

- Δυναμική προτεραιότητα (dynamic priority DP)

Τύπος: $DP = \text{MAX}[100, \{\min(SP - \text{bonus} + 5, 139)\}]$

Αν $PR < 120$ τότε έχουμε $(140 - PR) * 20\text{ms}$

π.χ. διεργασία με $PR = 100$ θα πάρει: $(140-100) * 20 = 800\text{ms}$

Αν $PR > 120$ τότε έχουμε $(140 - PR) * 5\text{ms}$

π.χ. διεργασία με $PR = 130$ θα πάρει: $(140-130) * 5 = 50\text{ms}$

π.χ. έστω μία διεργασία ξεκινάει με $SP = 105$ και λόγω “καλής συμπεριφοράς” παίρνει $\text{bonus} = 7$

άρα $DP = \text{MAX}[100, \{\min(105 - 7 + 5, 139)\}] =$
 $\text{MAX}[100, \{\min(103, 139)\}] =$
 $\text{MAX}[100, 103] = 103$

① Το bonus ορίζεται με βάση το SLEEP TIME

Δηλαδή SLEEP TIME = 0 - 99 ms, το bonus είναι 0

= 100 - 199 ms, το bonus είναι 1

= 200 - 299 ms, το bonus είναι 2

= ...

Παρατήρηση: Αν $\text{bonus} = 5$ τότε $DP = SP$

Αν $\text{bonus} > 5$ τότε $DP < SP$

Αν $\text{bonus} < 5$ τότε $DP > SP$

Μειονεκτήματα αυτού του χρονοδρομολογητή

- Διεργασίες με χαμηλό (όχι σε αριθμό) PR αργούν να τρέξουν
- Διεργασίες με μικρή διαφορά PR παίρνουν σημαντική διαφορά σε χρόνο

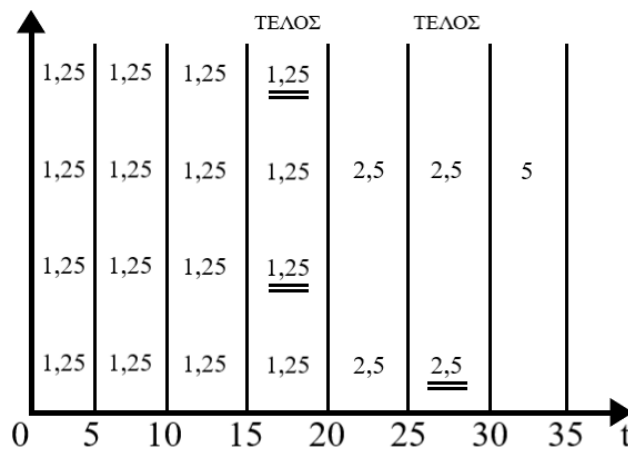
Completely Fair Scheduler (CFS)

4 διεργασίες με burst time (χρόνος χρήσης CPU)

Παράδειγμα 1	P0	10ms
	P1	5ms
	P2	20ms
	P3	5ms

ΙΔΑΝΙΚΗ ΚΑΤΑΣΤΑΣΗ(Virtual runtime
ιδεατός χρόνος εκτέλεσης)

- Βρίσκουμε το μικρότερο κβάντο (Vruntime) και χωρίζουμε το χρόνο σε τέτοια διαστήματα (στο παράδειγμα 1 το μικρότερο VR είναι 5ms)



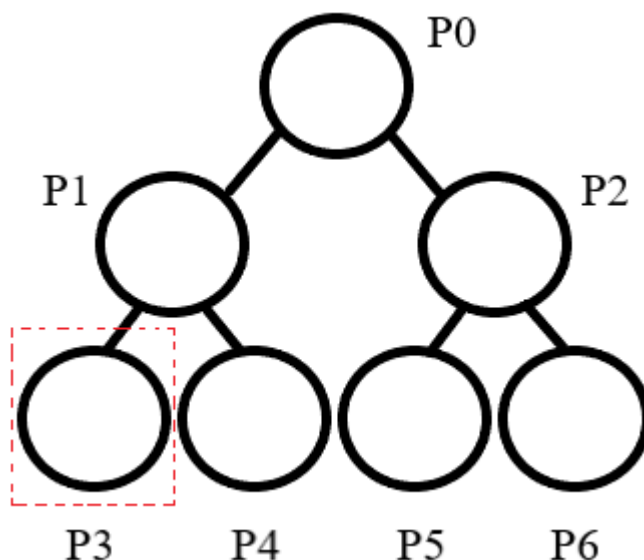
Υποθέτουμε ίδιο PRIORITY για τις 4 διεργασίες. Αν δεν ισχύει αυτό θέλουμε κάτι που να πλησιάζει σε αυτό.

$$VR = VR + (t * w)$$

σχέση 1

Αν κάποια διεργασία εκτελεστεί για κάποια κβάντα, το νέο της Vruntime θα είναι το προηγούμενο συν το $t * w$ (στάθμιση)

① Ο CFS διατηρεί ένα τέτοιο δέντρο:



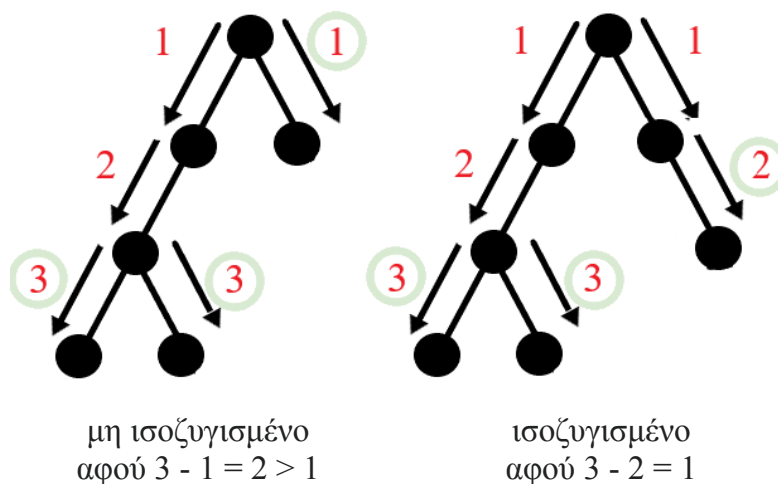
Οι κόμβοι (διεργασίες) τοποθετούνται στο δέντρο βάση Vruntime.

Στο παράδειγμα μας μικρότερο VR έχει η διεργασία P3.

- 1) Ξεκινάει η αριστερότερη (μικρότερο Vruntime)
- 2) Μόλις τελειώσει:
 - ή παραμένει στη θέση της (αν το VR της είναι ακόμα μικρότερο απ' όλα) και ξανά τρέχει
 - ή πάει δεξιά και έρχεται η επόμενη
- 3) Έστω ότι δεν έχουμε άλλες διεργασίες παρά αυτές. Μετά από ένα διάστημα θα τρέξουν όλες αφού πάνε δεξιά λόγω της σχέσης 1
- 4) Εισαγωγή νέας διεργασίας
 - a) Θα πάρει τόσο μικρό Vruntime ώστε να μπει αριστερά ($VR = 1$)
 - Πόσο χρόνο; Αρχικά όλες τρέχουν για προεπιλεγμένο χρόνο (τυπικά στο Linux) 4ms. Το νέο $VR = 4 + (4 * w)$
 - Πόσος χρόνο χρειάζεται για τον εντοπισμό του μικρότερου VR;

Balanced δέντρο:

η απόσταση από κάθε επίπεδο προς τα φύλλα δψεν μπορεί να διαφέρει πάνω από 1



Για να επεξεργαστούμε μία κορυφή ενός ισοζυγισμένου δέντρου με N κορυφές ο χρόνος είναι $\log_2(N)$ το πολύ.

nice: [-20...19]

$$w = 1024 / 1.25^{\text{nice}}$$

σχέση 2

Ενδεικτικά:	nice	w
	0	1024
	1	820
	-1	1277

Παρατηρούμε πως όσο το nice μειώνεται, το weight αυξάνεται και επομένως αυξάνεται και το ποσοστό χρόνου της CPU που θα δοθεί στη διεργασία.

TL = 20 msσχέση 3

Target Latency: ο ελάχιστος χρόνος που θα χρειαστεί για να πάρουν όλες οι διεργασίες με τη σειρά τους τη CPU.

MG = 4 msσχέση 4

Minimum Granularity: ο ελάχιστος χρόνος που θέλουμε να δίνουμε σε κάθε διεργασία.

- 1) Στον CFS το default TL = 20 ms. Αν έχω 5 διεργασίες, ποιο θα είναι το MG;
Οι 5 διεργασίες θα έχουν χρησιμοποιήσει με τη σειρά τους τη CPU σε TL αν MG = 4 ms
- 2) Έστω ότι N = 8. Ποια πολιτική πρέπει να ακολουθήσω ώστε να μην υπερφορτωθεί το σύστημα (δηλ. $N * MG > TL$);
Έχω δύο επιλογές:
 - 1.1) Αφήνω το MG σε default τιμή και ανεβάζω το TL σε $MG * N = 32$ ms
 - 1.2) Αφήνω το TL σε default τιμή και κατεβάζω το MG σε $TL / N = 2.5$ ms

① Επαναχρονοδρομολόγηση γίνεται δλδ ανά **$N * MG$ ms** σχέση 5

Rescheduling: κάθε $N * MG$ ο CFS βλέπει ποιες διαδικασίες είναι σε **HOLD** και τις βάζει αριστερά στο δέντρο με Vruntime = 1

Για κάθε διεργασία δίνουμε χρόνο: **$TL * (K / M)$**

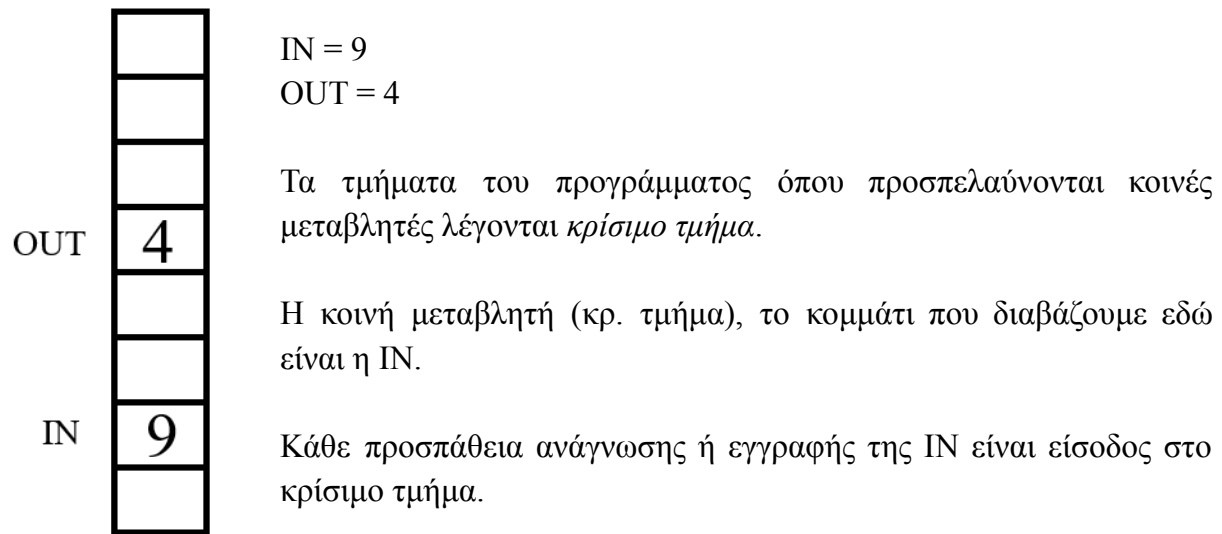
όπου $K = 1024 / 1.25^{\text{nice}}$ δηλαδή το weight της διεργασίας και

όπου $M = \sum_{i=1}^N (K_i)$ δηλαδή το weight της run queue

- ① K / M είναι μεταξύ (0, 1) άρα $TL * (K / M)$ είναι % του TL
CFS δίνει σε κάθε διεργασία ποσοστό του ανεκτού TL, γι' αυτό είναι FAIR

Στο spool directory υπάρχουν 2 μεταβλητές:

- **IN**: επόμενη ελεύθερη θέση
- **OUT**: διεργασία που εκτυπώνεται



ΒΗΜΑΤΑ

1. ΔΙΑΒΑΖΕΙ IN
2. ΓΡΑΦΕΙ ΤΟ ΟΝΟΜΑ ΑΡΧΕΙΟΥ ΣΤΗ ΘΕΣΗ SPOOL[IN]
3. $IN \leftarrow IN + 1$
4. ΕΚΤΥΠΩΣΗ

π.χ. Έστω 2 διεργασίες A, B όπου A θέλει να τυπώσει το αρχείο A.txt και B θέλει να τυπώσει το αρχείο B.txt

Αν το κομμάτι κώδικα στα αριστερά τρέξει για τη διεργασία A θα έχουμε:

- | | |
|--------------------------|--------------------------------|
| 1. MOVE <IN> <R1> | 1. $R1 \leftarrow 9$ |
| 2. MOVE <FILENAME> <R2> | 2. $R2 \leftarrow A.txt$ |
| 3. MOVE <R2> <SPOOL[R1]> | 3. $SPOOL[9] \leftarrow A.txt$ |
| 4. INC <R1> | 4. $R1 \leftarrow 10$ |
| 5. MOVE <R1> <IN> | 5. $IN \leftarrow 10$ |

Ερώτηση:

Έστω τρέχει η A και κόβεται μετά το τέλος του βήματος 1 και στη συνέχεια τρέχει η B και κόβεται μετά το τέλος του βήματος 3. Τι πρόβλημα θα δημιουργηθεί όταν ξανατρέξει η A;

Απάντηση:

Η A γράφει τη τιμή 9 στο χώρο μνήμης της και μπαίνει σε κατάσταση READY. Όταν ξανατρέξει, θα συνεχίσει από το βήμα 2.

Στη συνέχεια τρέχει η B:

1. $IN \leftarrow 9$ // αφού η A κόπηκε πριν αλλάξει το IN
2. $R2 \leftarrow B.txt$
3. $SPOOL[9] \leftarrow B.txt$

Στη συνέχεια τρέχει η A από το βήμα 2:

2. $R2 \leftarrow A.txt$
3. $SPOOL[9] \leftarrow A.txt$ // αφού $R1 = 9$
4. $R1 \leftarrow 10$
5. $IN \leftarrow 10$

Επομένως πλέον έχουμε χάσει τη B.

Χαρακτηριστικά που επιθυμούμε:

- Αμοιβαίος αποκλεισμός: μόνο μια διεργασία μπορεί να χρησιμοποιεί το Κ.Τ.
- Δεν μας ενδιαφέρει η ταχύτητα της CPU
- Μια διεργασία εκτός Κ.Τ. δεν πρέπει να μπλοκάρει τις άλλες από το να μουν σε Κ.Τ. Αν γίνει αυτό, αποκαλείται ΕΝΕΡΓΟΣ ΑΝΑΜΟΝΗ
- Είσοδος στο Κ.Τ. σε πεπρασμένο χρόνο

Αυστηρή εναλλαγή

(α)

1. while (TRUE){
2. while (turn != 0) // βρόγχος
3. critical_region(); // είσοδος σε ΚΤ
4. turn ← 1; // αλλαγή του turn ώστε η B να μπορεί να χρησιμοποιήσει το ΚΤ
5. noncritical_region(); //συνέχεια του προγράμματος
6. }

(β)

1. while (TRUE){
2. while (turn != 1)
3. critical_region();
4. turn ← 0;
5. noncritical_region();
6. }

① Στη γραμμή 2, ο βρόγχος επαναλαμβάνεται όσο το $turn \neq \#$. Η διεργασία χάνει κβάντα όσο αυτό γίνεται.

Θεωρούμε πως για το (α) το `noncritical_region()`; είναι λίγες απλές εντολές ενώ για το (β) είναι πολλές πολύπλοκες πράξεις.

π.χ. Ας πούμε πως η B εκτελεί ακόμα εντός στο `noncritical_region()`; ενώ η A έχει τυπώσει επομένως το `turn = 1`. Η B δεν είναι σε ΚΤ αλλά το `turn = 1` οπότε η A δεν μπορεί να χρησιμοποιήσει το ΚΤ (ΕΝΕΡΓΟΣ ΑΝΑΜΟΝΗ)

π.χ. Η B εκτελεί εντολές στο `noncritical_region()`; και η A αφού τυπώσει κάνει το `turn = 1`. Η A όμως έχει αρκετά κβάντα για να φτάσει πάλι σε σημείο όπου θέλει να χρησιμοποιήσει το ΚΤ αλλά δεν μπορεί αφού το `turn = 1` (ΕΝΕΡΓΟΣ ΑΝΑΜΟΝΗ)

TSL (Test and Set Lock)

- ```

1. enter_region: // προσπάθεια εισόδου σε ΚΤ
2. TSL REGISTER, LOCK // REG ← LOCK, LOCK ← 1
3. CMP REGISTER, #0 // Συγκρίνει REG ≠ 0
4. JNE enter_region // Jump Not Equal - αν REG ≠ 0 επαναλαμβάνεται η TSL
 - αν REG = 0 τότε μπαίνουμε σε ΚΤ
5. RET
6. leave_region:
7. MOVE LOCK, #0
8. RET

```

**π.χ.** Έστω ότι η Α μπαίνει σε ΚΤ και *δεν κόβονται τα κβάντα στο ΚΤ* και μετά προσπαθεί να μπει η Β. Δεν θα έχουμε πρόβλημα γιατί:

A:

2.  $TSL\ REG \leftarrow 0, LOCK \leftarrow 1$  //  $LOCK = 1$  σημαίνει πως κλείδωσε το ΚΤ
3. Συγκρίνει  $REG \neq 0$
4. Επειδή  $REG = 0$  μπαίνει στο ΚΤ

KT

5. RET επιστρέφουμε από το ΚΤ
6. LOCK  $\leftarrow 0$

Η Β βρίσκει LOCK = 0 και επαναλαμβάνει ότι η Α

**π.χ.** Έστω ότι η Α μπαίνει σε ΚΤ και *κόβονται τα κβάντα στο ΚΤ* πριν αυξηθεί η ΙΝ. Θα χαθεί εκτύπωση;

A:

2.  $TSL\ REG \leftarrow 0, LOCK \leftarrow 1$
3. Συγκρίνει  $REG \neq 0$
4. Επειδή  $REG = 0$  μπαίνει στο ΚΤ

```
R1 ← 7
R2 ← A.txt
SPOOL[7] ← A.txt
R1 ← 7 + 1
```

-κόβονται τα κβάντα-

## Context A

$R1 \leftarrow 8$  (η Α επιστρέφοντας θα θέσει  $IN \leftarrow R1$  άρα  $IN \leftarrow 8$ )

Αν η Β καταφέρει να μπει στο ΚΤ θα δει  $IN = 7$

B:

2. TSL REG  $\leftarrow$  1, LOCK  $\leftarrow$  1
3. Συγκρίνει REG  $\neq$  0
4. REG  $\neq$  0 άρα δεν μπαίνει στο KT έως ότου βγει η A και θέσει LOCK  $\leftarrow$  0

Επομένως δεν θα χαθεί εκτύπωση.

**π.χ.** Έστω η A εκτελείται, μπαίνει στο KT και μόλις ολοκληρώσει την εκτύπωση κόβεται, πριν κάνει LOCK  $\leftarrow$  0. Επίσης η A έχει μεγαλύτερη προτεραιότητα από τη B και επαναχρονοδρομολογείται (δεν έχουμε ενεργό αναμονή αφού η A θα ξανατρέξει και θα θέσει LOCK  $\leftarrow$  0. Η B όταν πάρει κβάντα θα μπει σε KT)

**π.χ.** Η B έχει μεγαλύτερη προτεραιότητα από την A και η A έχει κλειδώσει τη LOCK και έχει κοπεί πριν κάνει LOCK  $\leftarrow$  0. Η B θα τρέχει άσκοπα τη TSL πολλές φορές ενώ η A είναι εκτός KT (ΕΝΕΡΓΟΣ ΑΝΑΜΟΝΗ)

### Peterson

1. # define FALSE 0
2. # define TRUE 1
3. # define N 2
4. int turn; // δήλωση μεταβλητής turn
5. int interested[N]; // δήλωση μεταβλητής interested[N]
6. void enter\_region(int process){ // προσπάθεια εισόδου σε KT
7. int other; // τοπική μεταβλητή other που υποδυκνύει την “άλλη διεργασία”
8. other  $\leftarrow$  1 - process; // A = process = 0 όταν process = 0, other  $\leftarrow$  1  
B = process = 1 όταν process = 1, other  $\leftarrow$  0
9. interested[process]  $\leftarrow$  TRUE; // εκδήλωση ενδιαφέροντος για είσοδο σε KT π.χ.  
if p0 runs enter\_region it sets interested[0]  $\leftarrow$  1
10. turn  $\leftarrow$  process;
11. while (turn == process && interested[other] == TRUE)  
// είναι σειρά της διεργασίας με id = process να δοκιμάσει να μπει. Όσο η turn  
// έχει τη τιμή της διεργασίας που εκτελεί την enter\_region και ενδιαφέρεται η  
// άλλη, ΔΕΝ μπαίνει σε KT, δηλαδή επαναλαμβάνεται ο βρόγχος
12. }
13. void leave\_region(int process){
14. interested[process]  $\leftarrow$  FALSE;
15. }

**π.χ.** Έστω ότι μπαίνει η A, δεν κόβεται, τυπώνει και ακολουθεί η B. Υπάρχει πρόβλημα αν η A δεν κοπεί;

|            |   |   |
|------------|---|---|
| interested | 0 | 0 |
| turn       | 0 |   |
| IN         | 7 |   |

8. other  $\leftarrow$  1 - process = 1

9. interested[0]  $\leftarrow$  1

10. turn  $\leftarrow$  0

11. while (turn == process && interested[other] == 1)  
     0 == 0 TRUE                      interested[1] == 0 FALSE

|            |   |   |
|------------|---|---|
| interested | 1 | 0 |
| turn       | 0 |   |
| IN         | 7 |   |

Αρα η συνθήκη του while είναι ψευδής και η A μπαίνει σε ΚΤ επειδή η B δεν ενδιαφέρεται να μπει. Η A τυπώνει και θέτει interested[0]  $\leftarrow$  0

|            |   |   |
|------------|---|---|
| interested | 0 | 0 |
| turn       | 0 |   |
| IN         | 8 |   |

Ύπνος - Sleep

σε αναμονή

Μία διεργασία που δεν μπορεί να προχωρήσει σε ΚΤ, πηγαίνει σε κατάσταση **WAIT**, έως ότου να υπάρξει μια συνθήκη που θα την αφυπνήσει.

① Από μόνη της η χρήση της Ύπνωσης, δεν βοηθάει για να λυθούν όλα τα προβλήματα

**ΣΗΜΑΤΟΦΟΡΕΙΣ**

- 1) Ύπνωση
  - 2) Αδιαίρετο
- Κάθε σηματοφορέας εκτελεί αυτές τις δύο λειτουργίες

**DOWN(S):**

IF  $S > 0$  THEN

$S \leftarrow S - 1$

ELSE

SLEEP



Η διεργασία που κάλεσε το σηματοφορέα πάει για ύπνο και:

- 1) μπαίνει άλλη διεργασία για τρέξιμο άρα δεν ξοδεύεται CPU
- 2) η διεργασία γράφεται στη λίστα διεργασιών του σηματοφορέα (δηλ. ο MUTEX θα έχει λίστα με A, B, C και D)
- 3) η διεργασία πρέπει κάποια στιγμή να ξυπνήσει από μία συνθήκη

**UP(S):**

$S \leftarrow S + 1$

WAKEUP(Process\_ID)

Η UP δημιουργεί τη συνθήκη αφύπνισης. S μπορεί να έχει ελάχιστη τιμή 0 οπότε μία ή πολλές διεργασίες που θα προσπαθήσουν να μουν σε ΚΤ και θα βρουν  $S = 0$ , θα πάνε για ύπνο και κάποια στιγμή όταν  $S = 1$  ( $S \leftarrow S + 1$ ) μία από αυτές θα ξυπνήσει.

① Η DOWN(S) και η UP(S) είναι αδιαίρετες

① Η MUTEX είναι κοινή μεταβλητή, αρχικά = 1

αμοιβαίος αποκλεισμός

**π.χ.** Να δείξετε ότι με χρήση ενός σηματοφορέα MUTEX (mutual exclusion) επιλύεται χωρίς προβλήματα το ζήτημα των εκτυπώσεων μεταξύ 4 διεργασιών

Δομή:

**DOWN(MUTEX)**

1...5 ΚΤ

**UP(MUTEX)**

- Έστω ότι τρέχει πρώτα η A
- Το MUTEX αρχικά είναι 1
- Καλείται η DOWN:  $MUTEX = 1 > 0$  άρα  $MUTEX \leftarrow MUTEX - 1 = 0$
- Έστω ότι η A κόβεται στη γραμμή 3 του KT
- Έστω πως επόμενη τρέχει η C
- Καλείται η DOWN:  $MUTEX == 0$  άρα SLEEP
- Η C μπαίνει σε Ύπνωση, δε καταναλώνει χρόνο CPU άρα δεν υπάρχει ενεργός αναμονή. MUTEX
- Γίνονται τα ίδια με D και B και έχουμε: MUTEX

① Στη λίστα MUTEX βρίσκονται όλες οι διεργασίες που έχουν πέσει για ύπνο εξαιτίας του σηματοφορέα MUTEX

- Τρέχει ξανά η A 

|   |  |  |
|---|--|--|
| C |  |  |
|---|--|--|
- A ολοκληρώνει το KT 

|   |   |   |
|---|---|---|
| C | D | B |
|---|---|---|
- Καλείται η UP:  $S \leftarrow S + 1 = 1$ , WAKEUP(Process\_ID)
  - ή θα ξυπνήσει μία
  - ή θα ξυπνήσουν όλες
    - Οι διεργασίες C, D, B θα γίνουν **READY** άρα μπορούν να χρονοδρομολογηθούν

Producer: Συλλέγει και γράφει δεδομένα σε μία κοινή μνήμη

Consumer: Διαβάζει δεδομένα αφαιρώντας τα από τη κοινή μνήμη, και τα επεξεργάζεται

### ΛΥΣΗ ΧΩΡΙΣ ΣΗΜΑΤΟΦΟΡΕΙΣ

#### Producer

1. `int item;` // το στοιχείο που προστίθεται στη μνήμη
2. `while (TRUE){`
3. `item ← produce_item();` // ανάγνωση ενός στοιχείου
4. `if (count == N) sleep();` // μνήμη έχει N capac., αν γεμίσει ο produ. πάει για ύπνο
5. `insert_item(item);` // γράφουμε στη μνήμη
6. `count ← count + 1;` // αυξάνεται το count
7. `if (count == 1) wakeup(consumer);` // αν count == 0 (κενή μνήμη) τότε ο consu. πάει για ύπνο αφού δεν υπάρχει κάτι για να καταναλώσει. Μόλις count == 1, ξυπνάμε τον consu. αφού είχε πάει για ύπνο αφού ήταν count == 0
8. `}`

Consumer

1. `int item;` // το στοιχείο που θα αφαιρείται από τη μνήμη
2. `while(TRUE){`
3.   `if (count == 0) sleep();` // αν η μνήμη είναι κενή, ο καταναλωτής πάει για ύπνο
4.   `item ← remove_item();` // διαβάσει το στοιχείο και το αφαιρεί από τη μνήμη
5.   `count ← count - 1;` // μειώνεται το count
6.   `if (count == N - 1) wakeup(producer);` // αν υπάρχει κενή θέση, ο παραγωγός μπορεί να κάνει κάτι. Η προηγούμενη τιμή count ήταν N πριν την αφαίρεση άρα ο παραγωγός πήγε γι ύπνο
7.   `consume_item(item);` // καταναλώνεται το στοιχείο
8. `}`

επομένως:

Producer:

Κοιμάται αν βρει `count == N` (γραμμή 4 producer)

Ξυπνάει αν `count == N - 1` (γραμμή 6 consumer)

Αυτό συμβαίνει όταν η μνήμη έχει γεμίσει (πάει για ύπνο) και ο consumer καταναλώσει ένα στοιχείο αφήνοντας το `count == N - 1`

Consumer:

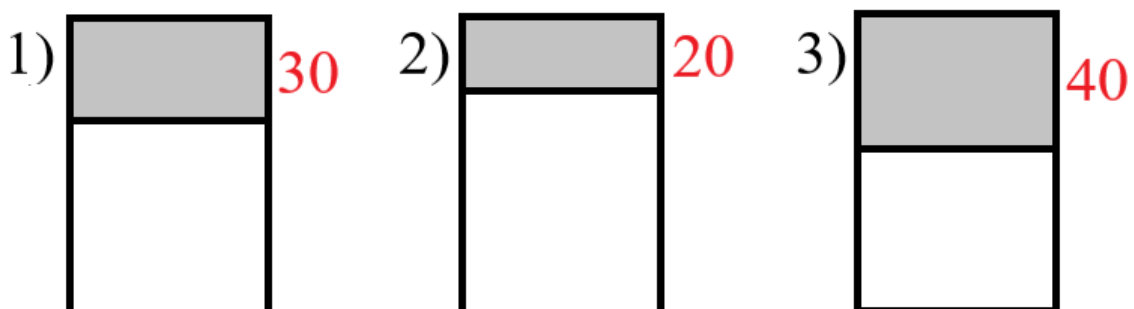
Κοιμάται αν βρει `count == 0` (γραμμή 3 consumer)

Ξυπνάει αν `count == 1` (γραμμή 7 producer)

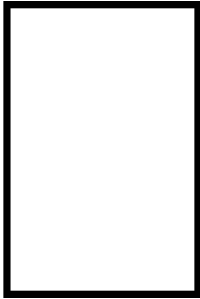
Αυτό συμβαίνει όταν η μνήμη έχει αδειάσει (πάει για ύπνο) και ο producer προσθέτει ένα στοιχείο αφήνοντας το `count == 1`

Λαμβάνοντας υπ' όψιν ότι δεν υπάρχει αδιαίρετο, εξηγήστε τι θα συμβεί με την ακόλουθη σειρά γεγονότων:

- 1) Ο παραγωγός τοποθετεί 30 στοιχεία στη κοινή μνήμη, κόβεται
- 2) Ο καταναλωτής καταναλώνει 10 στοιχεία και κόβεται
- 3) Ο παραγωγός τοποθετεί 20 στοιχεία
- 4) Ο καταναλωτής διαβάζει όλα τα στοιχεία και κόβεται όταν διαβάσει το count
- 5) Ο παραγωγός τοποθετεί 1 στοιχείο
- 6) Ο καταναλωτής επιστρέφει
- 7) Ο παραγωγός γεμίζει τη μνήμη



4)



Ο καταναλωτής διαβάζει τη  $count == 0$  αλλά δεν πάει για ύπνο αφού κόβεται, δηλαδή δεν προλαβαίνει να στείλει σήμα ύπνωσης

5)



Η γραμμή 7 του producer σημαίνει ότι στέλνουμε σήμα προς το ΛΣ να αφυπνίσει τον consumer (ο οποίος όμως δεν κοιμάται). Το σήμα αφύπνισης χάνεται

6) Ο consumer φέρνει μαζί το CONTEXT,  $count == 0$ , στέλνει σήμα ύπνωσης και πάει για ύπνο.

7) Ο producer γεμίζει τη μνήμη,  $COUNT == 100$  (έστω  $N == 100$ ) και πάει για ύπνο.

Producer: Συλλέγει και γράφει δεδομένα σε μία κοινή μνήμη

Consumer: Διαβάζει δεδομένα αφαιρώντας τα από τη κοινή μνήμη, και τα επεξεργάζεται

### ΛΥΣΗ ΜΕ ΣΗΜΑΤΟΦΟΡΕΙΣ

MUTEX → Για είσοδο στη κοινή μνήμη

EMPTY → Αρχικά ίσος με N (capacity) και αν φτάσει  $EMPTY == 0$  τότε ο producer πάει για ύπνο.

#### Producer

1. int item;
2. while (TRUE){
3.   item ← produce\_item();
4.   DOWN(EMPTY); // αν οι κενές θέσεις είναι 0 πάει για ύπνο
5.   DOWN(MUTEX); // locks memory access, κατεβάζοντας το MUTEX
6.   insert\_item(item);
7.   UP(MUTEX); // unlocks memory access, ανεβάζοντας το MUTEX
8.   UP(FULL); // ο σηματοφορέας FULL αυξάνεται κατά 1 και ξυπνάει ο consumer
9. }

#### Consumer

1. int item;
2. while (TRUE){
3.   DOWN(FULL); // αν οι γεμάτες θέσεις είναι 0 πάει για ύπνο
4.   DOWN(MUTEX); // locks memory access, κατεβάζοντας το MUTEX
5.   item ← remove\_item();
6.   UP(MUTEX); // unlocks memory access, ανεβάζοντας το MUTEX
7.   UP(EMPTY); // ο σηματοφορέας EMPTY αυξάνεται κατά 1 και ξυπνάει ο prod.
8.   consume\_item(item);
9. }

① Οι γραμμές 4, 5, 7 και 8 του producer, και 3, 4, 6 και 7 του consumer, είναι **αδιαίρετες**, δηλαδή δεν μπορούν να κοπούν λόγω κβάντων, παρά μόνο μετά το τέλος τους.

### Ερώτημα 1/2

Έστω ότι ο παραγωγός ξεκινά και προσθέτει 30 στοιχεία και κόβεται μόλις ανεβάσει το MUTEX. Πόσα στοιχεία το πολύ μπορεί να διαβάσει ο καταναλωτής αν δεν τρέξει ξανά ο παραγωγός; (Αρχικά  $MUTEX == 1$ ,  $EMPTY == 100$ ,  $FULL == 0$ )

*1η επανάληψη*

- empty θα γίνει 99 (line 4)
- mutex θα γίνει 0 (line 5)



- mutex θα γίνει 1 (line 7)
- full θα γίνει 1 (line 8)

*2η επανάληψη*

- empty θα γίνει 98 (line 4)
- mutex θα γίνει 0 (line 5)
- mutex θα γίνει 1 (line 7)
- full θα γίνει 2 (line 8)

*30η επανάληψη*

- empty θα γίνει 70 (line 4)
- mutex θα γίνει 0 (line 5)
- mutex θα γίνει 1 (line 7) και *κόβεται* ενώ full == 29

Ο καταναλωτής μπορεί να διαβάσει 29 στοιχεία άσχετα αν έχουν γραφτεί 30. Ο καταναλωτής θα μειώσει το full έως ότου γίνει 0.

**Ερώτημα 2/2**

Έστω ότι ο καταναλωτής προλαβαίνει να διαβάσει 25 στοιχεία και κόβεται μόλις ανεβάσει το EMPTY. Ποιες θα είναι οι τιμές των σηματοφορέων;

*1η επανάληψη*

- full θα γίνει 28 (line 3)
- mutex θα γίνει 0 (line 4)
- mutex θα γίνει 1 (line 6)
- empty θα γίνει 71 (line 7)

*2η επανάληψη*

- full θα γίνει 27 (line 3)
- mutex θα γίνει 0 (line 4)
- mutex θα γίνει 1 (line 6)
- empty θα γίνει 72 (line 7)

*25η επανάληψη*

- full θα γίνει 4 (line 3)
- mutex θα γίνει 0 (line 4)
- mutex θα γίνει 1 (line 6)
- empty θα γίνει 95 (line 7)

Ασκήσεις

- 1) Έστω ότι ξεκινάει ο παραγωγός και κόβεται κατά τη διαδικασία εγγραφής στοιχείου (γραμμή 6)
- 2) Έστω ότι κόβεται μόλις ανεβάσει το MUTEX (γραμμή 7)
- 3) Έστω ότι κόβεται μόλις ανεβάσει το FULL (στη περίπτωση αυτή εκτελείται ο καταναλωτής)

Θα υπάρξει σε κάποια από αυτές τις περιπτώσεις ενεργός αναμονή, επιτυγχάνεται αμοιβαίος αποκλεισμός;

Λύσεις

- 1) Έστω ότι τα κβάντα κόβονται όταν διαβαστεί από τον MAR, η διεύθυνση μνήμης = 1000.

|      |  |
|------|--|
| 1000 |  |
| 1099 |  |

Ο παραγωγός κάνει context switch και αποθηκεύει τον MAR στο χώρο μνήμης της διεργασίας.

EMPTY == 99 (line 4)  
MUTEX == 0 (line 5)

Έρχεται ο καταναλωτής και αφού FULL == 0 (αφού ο παραγωγός δεν έχει εκτελέσει τη γραμμή 8) ο καταναλωτής γράφεται στη λίστα του σηματοφορέα FULL και πάει για ύπνο. Επομένως δεν υπάρχει ενεργός αναμονή κι επιτυγχάνεται αμοιβαίος αποκλεισμός.

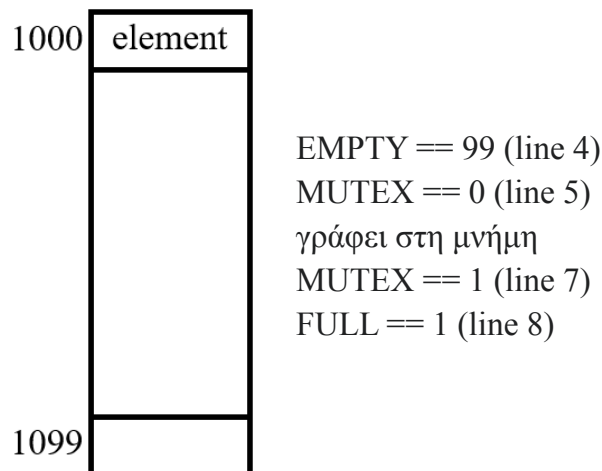
- 2) Έστω ότι τα κβάντα κόβονται μετά το UP(MUTEX)

|      |         |
|------|---------|
| 1000 | element |
| 1099 |         |

EMPTY == 99 (line 4)  
MUTEX == 0 (line 5)  
γράφει στη μνήμη  
MUTEX == 1 (line 7)

Έρχεται ο καταναλωτής και αφού FULL == 0 (αφού ο παραγωγός δεν έχει εκτελέσει τη γραμμή 8) ο καταναλωτής γράφεται στη λίστα του σηματοφορέα FULL και πάει για ύπνο. Επομένως δεν υπάρχει ενεργός αναμονή κι επιτυγχάνεται αμοιβαίος αποκλεισμός.

3) Έστω ότι τα κβάντα κόβονται μετά το UP(FULL)



Έρχεται ο καταναλωτής και κάνει:

*1η επανάληψη*

FULL == 0 (line 3)

MUTEX == 0 (line 4)

διαβάζει το στοιχείο

MUTEX == 1

EMPTY == 100

*2η επανάληψη*

αφού FULL == 0 ο καταναλωτής γράφεται στη λίστα του σηματοφορέα FULL και πάει για ύπνο

Ο παραγωγός είναι ενεργός αφού EMPTY > 0 άρα μπορεί να τρέξει και να βάλει στοιχεία. Ο καταναλωτής θα ξυπνήσει ξανά όταν τρέξει ο παραγωγός και προσθέσει ένα στοιχείο.

### Άσκηση

Μία εφαρμογή χρησιμοποιεί 3 δ. παραγωγών και 2 καταναλωτών. Οι παραγωγοί γράφουν στη μνήμη με τη σειρά P1, P2, P3 και οι καταναλωτές καταναλώνουν όλα τα στοιχεία με τυχαία σειρά. Να ορίσετε τους σηματοφορείς ώστε η εφαρμογή να δουλεύει σωστά. Εξηγήστε την απάντησή σας. (δεν μας ενδιαφέρει πόσα στοιχεία γράφονται, ότι γράφεται καταναλώνεται)

### Λύση

P1, P2, P3, C1, C2 (σηματοφορείς)

Εφ' όσον ξεκινάει ο P1 = 1, P2 = P3 = C1 = C2 = 0

P1

down(P1)

down(P1)

insert(ITEMS)

UP(P2)

Η P1 όπως και όλες ελέγχουν αν πρέπει να πάει για ύπνο. Αν όχι τότε προσθέτει στοιχεία και ξυπνά την επόμενη. (με το insert εισάγονται όσα στοιχεία πρέπει να εισαχθούν)

P2

down(P2)

insert(ITEMS)

UP(P3)

Οι P2 και P3 ακόμα και αν δρομολογηθούν πριν την P1 θα κάνουν down τον σηματοφορέα τους και θα πάνε για ύπνο.

P3

down(P3)

insert(ITEMS)

UP(C1)

UP(C2)

Η P3 “ξυπνάει” και τους 2 καταναλωτές, το ότι μία δ. ξυπνάει πρώτη δεν σημαίνει ότι θα τρέξει πρώτη

Το ποια από τις C1, C2 θα τρέξει καθορίζεται από το ΛΣ.

C1

down(C1)

consume(ITEMS)

UP(P1)

C2

down(C2)

consume(ITEMS)

UP(P1)

Αν τρέξει ο C1 και κάνει  $P1 = 1$  μπορεί να ξεκινήσει ο P1 πριν τον C2 άρα δεν επιβάλουμε σειρά P1, P2, P3, C1, C2

Η λύση είναι να βάλω ένα ακόμα down(P1)

Έστω ότι ξεκινάει η εφαρμογή  $P1 = 2, P2 = P3 = C1 = C2 = 0$  θα έχουμε:

P1

down(P1)

down(P1)

insert(ITEMS)

UP(P2)

P1

 $P1 = 1$  $P1 = 0$ 

εισαγωγή στοιχείων

 $P2 = 1$ 

Όταν φτάσουμε στους καταναλωτές

C1

down(C1)

consume(ITEMS)

UP(P1)

C1

 $C1 = 0$ 

κατανάλωση στοιχείων

 $P1 = 1$ 

Αν πάει να τρέξει ο P1 στο πρώτο down(P1) θα γίνει  $P1 = 0$  και στο δεύτερο down(P1) θα πάει για ύπνο. Για να ξυπνήσει ο P1 θα πρέπει να εκτελεστεί ο C2 να κάνει το  $P1 = 1$

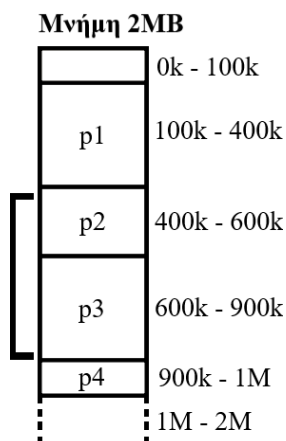
Κατανομή μνήμης

Δυναμική

Στατική

- 1) Δυναμική κατανομή: λαμβάνει υπ' όψιν τα μεγέθη των διεργασιών
- 2) Κατακερματισμός: Εσωτερικός  
Σημαίνει ότι η μνήμη ανήκει στο ΛΣ αλλά παραμένει αχρησιμοποίητη
- 3) Κατακερματισμός: Εξωτερικός  
Σημαίνει ότι η μνήμη ανήκει στη διεργασία αλλά παραμένει αχρησιμοποίητη  
π.χ. διεργασία με 7κ: παίρνει 2 σελίδες 4κ και το 1κ κατακερματίζεται
- 4) Στατική κατανομή
  - a) Πόσος χρόνος χρειάζεται για το ΛΣ ώστε να κατανέμει τη μνήμη για τις διεργασίες
  - b) Προβλήματα εξωτερικού κατακερματισμού
  - c) Αποτελεσματικότητα

π.χ.



i) Έστω ότι ολοκληρώνονται (φεύγουν) οι δ. P2, P3. Πως θα τοποθετηθούν οι δ. P5, P6, P7 που έρχονται, με τον αλγόριθμο Best-Fit;

P5 = 250k  
P6 = 300k  
P7 = 400k

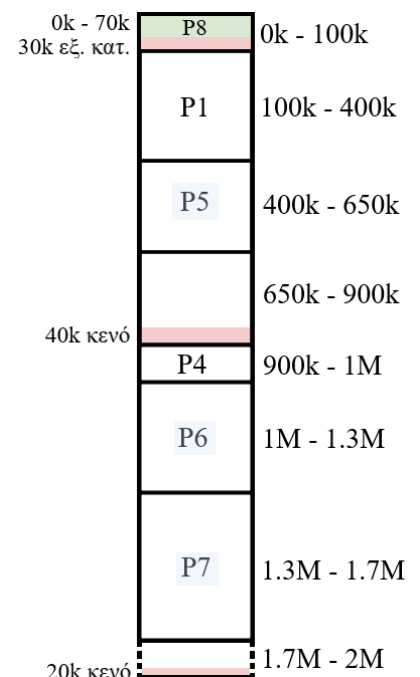
**Best-Fit:** για κάθε νέα δ. ψάχνω το “μικρότερο” χώρο όπου μπορεί να τοποθετηθεί.

Ανάλυση για P5 (το ίδιο ισχύει και για τις άλλες):

Η P5 δεν μπορεί να μπει στο κενό 0k - 100k. Μπορεί να μπει στο κενό 400k - 900k και θα δημιουργήσει κενό 500k - 250k = 250k, και στο κενό 1M - 2M όπου θα αφήσει κενό 1M - 250k = 750k

ii) Έστω ότι έρχεται η P8 = 70k και δεν έρχεται ξανά δ < 30k.

Αν έρχονται συνεχώς δ. των 70k θα δημιουργήσουν **κενά**

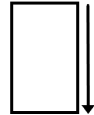


**First-Fit:** η δ. μπαίνει στο πρώτο κενό. Πιο γρήγορη από τη best αλλά προφανώς μπορεί να δημιουργήσει ακόμα μεγαλύτερο κατακερματισμό.

① Στο συγκεκριμένα παράδειγμα τυγχάνει να έχει τον ίδιο πίνακα με τη Best-Fit

**Worst-Fit:** η δ. μπαίνει στο μεγαλύτερο κενό. Αν μια δ. μπει στο μεγαλύτερο κενό θα υπάρχει αρκετός χώρος για να μπουν και άλλες

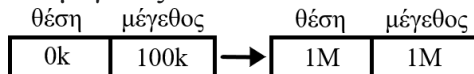
**Next-Fit:** η αναζήτηση για κενές θέσεις συνεχίζεται από το σημείο που εντοπίστηκε το τελευταίο κενό. δηλαδή γεμίζει έτσι:



Έστω ότι έχω τον αρχικό πίνακα:

Κάθε στοιχείο της λίστας πρέπει να κρατάει 2 τιμές:

- 1) Από που αρχίζει το κενό
- 2) Το μέγεθος του κενού

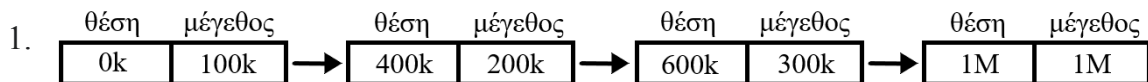


#### Ερώτηση:

Φεύγουν από τη λίστα οι P2, P3:

1. Να προστεθούν στη λίστα τα κενά
2. Να ενωθούν τα κενά

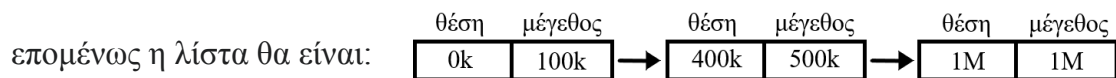
#### Απάντηση:



2. Αν  $\text{θέση}[i] + \text{μέγεθος}[i] = \text{θέση}[i+1]$ , ένωσε τα κενά

π.χ.  $400 + 200 = 600$

άρα  $\text{μέγεθος}[i] \leftarrow \text{μέγεθος}[i] + \text{μέγεθος}[i+1]$  και  $i.\text{next} \leftarrow (i+1).\text{next}$



#### Ερώτηση:

Πόσος χρόνος χρειάζεται θεωρητικά κατά μέση περίπτωση για να εκχωρήσουμε μνήμη σε μία διεργασία;

#### Απάντηση:

Κάθε δ. που λαμβάνει χώρο στη μνήμη μετά από κάποιες εκχωρήσεις, τοποθετεί μια εγγραφή στη συνδ. λίστα. Μια μέση περίπτωση, λέει ότι θα δημιουργηθούν K κενά, όπου  $K = N / 2$  ( $N$  = πλήθος διερ.)

Άρα για κάθε δ. πρέπει να γίνουν κατά μέσο όρο  $N / 2$  συγκρίσεις με τιμές τις λίστες για να βρεθεί το κενό.

1η διερ.      0 συγκρ.  
 2η διερ.       $2/2 = 1$  συγκρ.  
 3η διερ.       $3/2 = 1$  ή 2 συγκρ.

Αν προσθέσω αυτές τις εγγραφές:

$$\frac{1}{2} + \frac{2}{2} + \frac{3}{2} + \dots + \frac{\kappa-2}{2} + \frac{\kappa-1}{2} =$$

$$\left(\frac{1}{2} + \frac{\kappa-1}{2}\right) + \left(\frac{2}{2} + \frac{\kappa-2}{2}\right) + \left(\frac{3}{2} + \frac{\kappa-3}{2}\right) =$$

$$\frac{\kappa}{2} + \frac{\kappa}{2} + \frac{\kappa}{2} + \dots + \frac{\kappa}{2} = \frac{\kappa}{2} \times \frac{\kappa}{2} = \frac{\kappa^2}{4}$$

### Παράδειγμα

Μνήμη 1M και τα εξής γεγονότα      Να δείξετε την κατανομή με την **μέθοδο των φίλων**

A: 70k

B: 35k

C: 80k

Τέλος A

D: 60k

Τέλος B

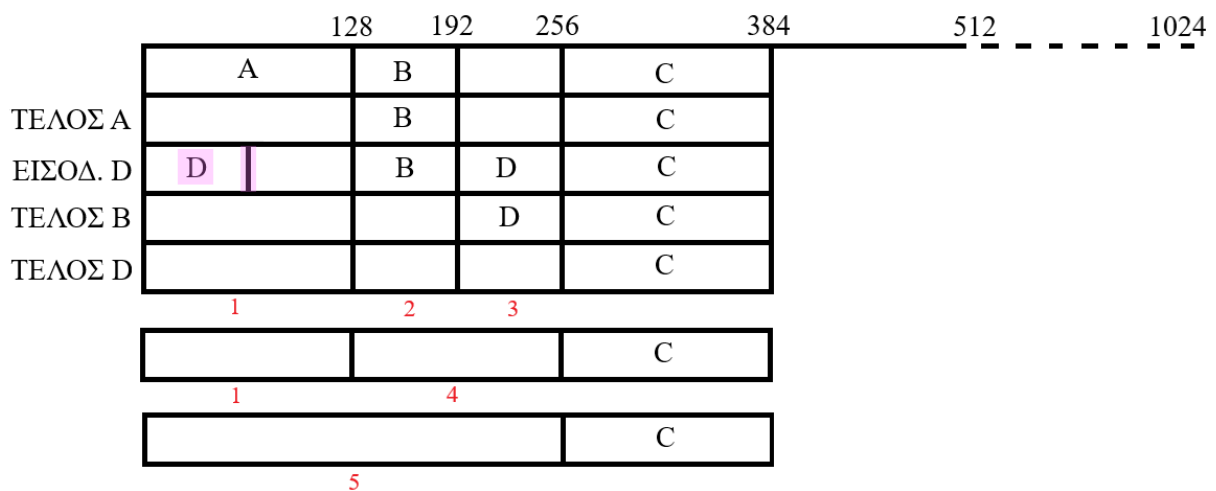
Τέλος D

Η A έχει κατακερματισμό  $128 - 70 = 58k$

Η B έχει κατακερματισμό  $64 - 35 = 29k$

Η C έχει κατακερματισμό  $128 - 80 = 48k$

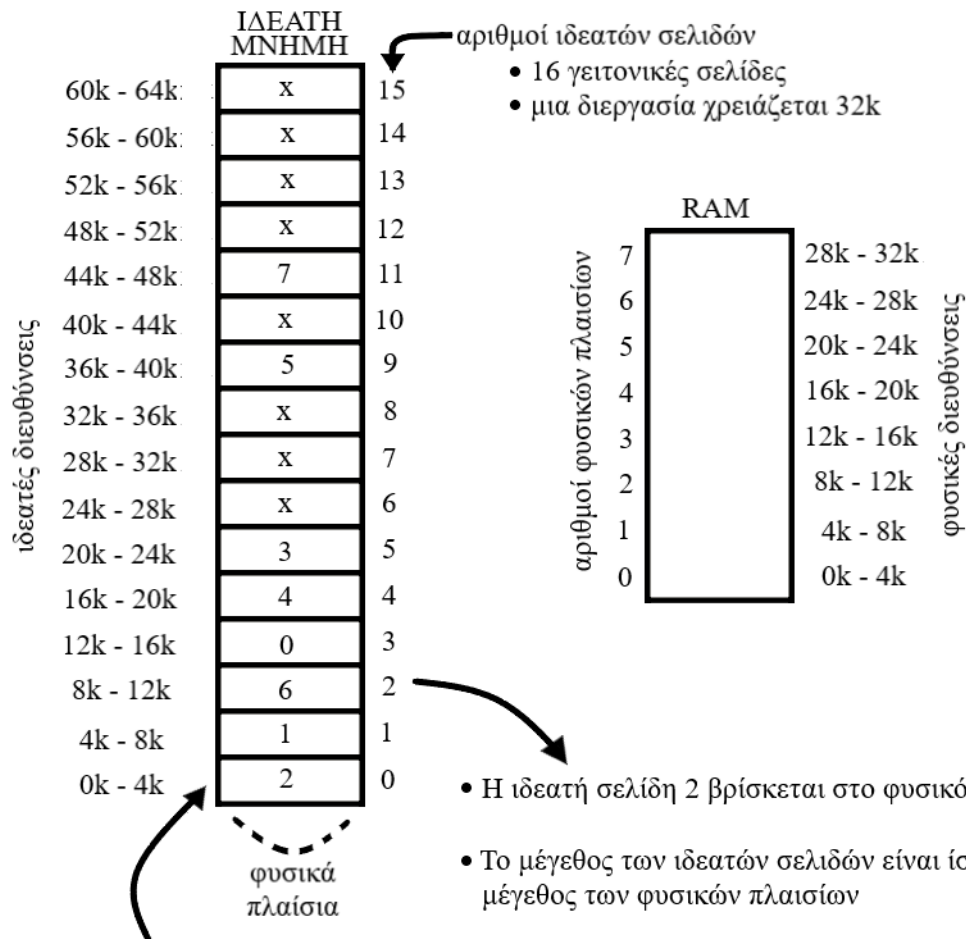
Η D έχει κατακερματισμό  $64 - 60 = 4k$



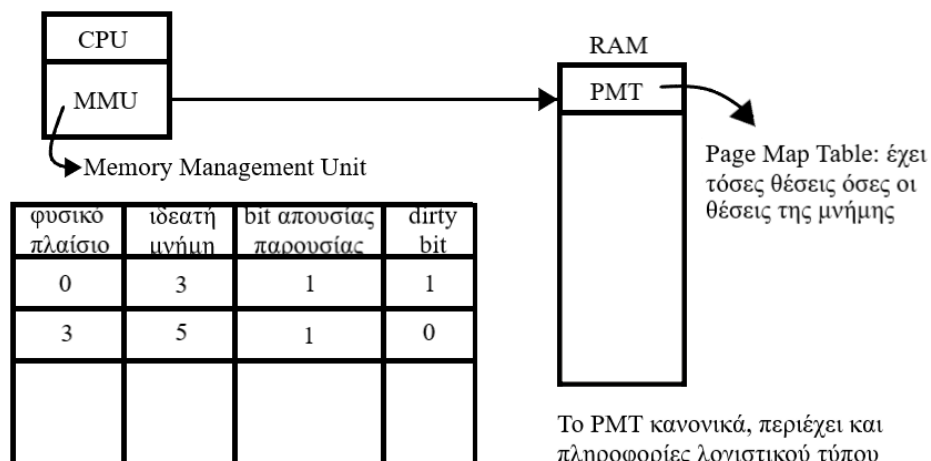
Το 1 και 2 δεν είναι φίλοι γιατί δεν έχουν το ίδιο μέγεθος, άρα δεν ενώνονται. Το 2 και 3 είναι φίλοι (2 των 64) άρα ενώνονται στο "4". Το 4 και το 1 είναι φίλοι (2 των 128) άρα ενώνονται στο "5".

**Buddy System in Linux → Αντιμετώπιση Κατακερματισμού και Slab Allocator**

- Σελιδοποίηση: η μνήμη διαιρείται σε μικρά τμήματα, τα οποία ονομάζονται σελίδες (τυπικά, κάθε σελίδα είναι 4kB). Μια διεργασία που ζητάει xkB θα πάρει xkB/4kB σελίδες



- Κάθε νούμερο συνοδεύεται με το Process ID της διεργασίας του
- Η σελιδοποίηση είναι η βάση της εικονικής μνήμης (virtual memory)
- Μια διεργασία αποθηκεύεται στο δίσκο. Το πρόγραμμα που έχει γραφεί, ζητάει δεδομένα που βρίσκονται στο δίσκο. Όμως για να τρέξει το πρόγραμμα γρήγορα, πρέπει οι ιδεατές διευθύνσεις δίσκου να μετατραπούν σε φυσικές.
- Η CPU ζητάει τη φυσική διεύθυνση που βρίσκεται μια ιδεατή σελίδα ώστε να την προσπελάσει από τη μνήμη. (π.χ. τα δεδομένα της ιδεατής σελίδας 0 που βρίσκονται στη φυσική σελίδα 2, φορτώνονται στη μνήμη)





Η CPU διαβάζει και ζητάει ιδεατές διευθύνσεις από το MMU. Το MMU επικοινωνεί με το PMT και διαβάζει εάν υπάρχει η ιδεατή σελίδα και που βρίσκεται στη φυσική μνήμη. Η CPU μαθαίνοντας αυτή τη πληροφορία, ζητάει πλέον τις φυσικές διευθύνσεις.

### Παράδειγμα

Η CPU ζητάει τη διεύθυνση 4090. Να περιγράψετε τη διαδικασία εύρεσης της αντίστοιχης φυσικής διεύθυνσης.

Ιδεατή διεύθυνση:

|            |                 |
|------------|-----------------|
| 4bit<br>PN | 12bit<br>OFFSET |
|------------|-----------------|

- Το PN (*page number*) εξαρτάται από το πλήθος των σελίδων. Π.χ. αν έχουμε μνήμη 64k / 4k = 16 σελίδες. Επομένως απαιτούνται  $\log_2(16) = 4\text{bit}$  για διευθυνσιοδότηση κάθε ιδεατής σελίδας

Γενικότερα: για K ιδεατές σελίδες χρειαζόμαστε  $\log_2(K)\text{bit}$  για το PN

- Το OFFSET διευθυνσιοδοτεί byte. Για σελίδες 4k χρειάζεται  $\log_2(4k) = 12\text{bit}$

Γενικότερα: για B bytes χρειαζόμαστε  $\log_2(B)\text{bit}$

|      |             |
|------|-------------|
| 4bit | 12bit       |
| 0000 | 11111111010 |

Η CPU παίρνει το κομμάτι PN και το δίνει στο MMU

| φυσικό πλαίσιο | ιδεατή μνήμη | bit απουσίας παρουσίας | dirty bit |
|----------------|--------------|------------------------|-----------|
| 2              | 0            | 1                      |           |
| 1              | 1            | 1                      |           |
| 6              | 2            | 1                      |           |

Η ιδεατή σελίδα 0 (0000) βρίσκεται στο φυσικό πλαίσιο 2. Αυτή η πληροφορία δίνεται από το MMU στη CPU.

Ιδεατή σελίδα 0

|      |      |
|------|------|
| 4090 | 4096 |
|------|------|

Η διεύθυνση 4090 είναι το byte 4090 της ιδεατής μνήμης 0.

Στο φυσικό πλαίσιο 2, που βρίσκεται το byte 4090;

|            |                 |
|------------|-----------------|
| 3bit<br>PN | 12bit<br>OFFSET |
|------------|-----------------|

Το OFFSET είναι το ίδιο με της ιδεατής μνήμης.

Το PN εξαρτάται από το πλήθος των φυσικών πλαισίων ( $32 / 4 = 8$ ). Επομένως απαιτούνται  $\log_2(8) = 3\text{bit}$  για διευθυνσιοδότηση κάθε φυσικού πλαισίου.

Συνεπώς η φυσική διεύθυνση είναι:

|      |             |
|------|-------------|
| 3bit | 12bit       |
| 010  | 11111111010 |

→ 12282 (8192 + 4090)

- 1) Η CPU ζήτησε την ιδεατή διεύθυνση 4090 από το MMU
- 2) Το MMU πήρε το πεδίο PN = 0000 της ιδεατής διεύθυνσης και το έψαξε στο PMT (η 1η ανάγνωση γίνεται από την RAM(MMU))
- 3) Η καταχώρηση του PMT για PN = 0000 είναι 2
- 4) Η τιμή 2 γράφεται με 3bit στη φυσική διεύθυνση 010
- 5) Σχηματίστηκε φυσική διεύθυνση 0101111111010 = 12282 (δεδομένα)
- 6) Η τιμή 12282 δόθηκε στον MAR (2η ανάγνωση από MAR(RAM))
- 7) Ανάγνωση της λέξης 12282 από τη RAM

Συνολικά έχουμε 2 αναγνώσεις από τη μνήμη.

### Παράδειγμα

Έστω ότι το **TLB (Translation Lookaside Buffer)** αποθηκεύει τις 3 πρώτες γραμμές του PMT. Κάθε ανάγνωση μνήμης απαιτεί 10 χρονικές μονάδες.

Να δώσετε τη φυσική διεύθυνση, στην οποία αποθηκεύεται η ιδεατή διεύθυνση 8190 και να βρείτε πόσο χρόνο απαιτεί η ανάγνωση της λέξης

| 20k - 24k | 3           | 5 | <table border="1"> <thead> <tr> <th>PN</th> <th>OFFSET</th> </tr> </thead> <tbody> <tr> <td>0001</td> <td>11111111010</td> </tr> </tbody> </table> <p>Η CPU ελέγχει πρώτα το TLB. Στέλνει τον αριθμό<br/>ιδεατής σελίδας 0001 στο TLB. Επειδή η<br/>καταχώρηση της ιδεατής σελίδας 1 υπάρχει στο<br/>TLB, δεν χρειάζεται ανάγνωση από τη μνήμη (PMT)</p> | PN | OFFSET | 0001 | 11111111010 |
|-----------|-------------|---|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|--------|------|-------------|
| PN        | OFFSET      |   |                                                                                                                                                                                                                                                                                                                                                          |    |        |      |             |
| 0001      | 11111111010 |   |                                                                                                                                                                                                                                                                                                                                                          |    |        |      |             |
| 16k - 20k | 4           | 4 |                                                                                                                                                                                                                                                                                                                                                          |    |        |      |             |
| 12k - 16k | 0           | 3 |                                                                                                                                                                                                                                                                                                                                                          |    |        |      |             |
| 8k - 12k  | 6           | 2 |                                                                                                                                                                                                                                                                                                                                                          |    |        |      |             |
| 4k - 8k   | 1           | 1 |                                                                                                                                                                                                                                                                                                                                                          |    |        |      |             |
| 0k - 4k   | 2           | 0 |                                                                                                                                                                                                                                                                                                                                                          |    |        |      |             |

| 3bit | 12bit       |
|------|-------------|
| 001  | 11111111010 |

 12-14bit 8-12bit  
 → 12282 (8190 + 4096)
 

Αν δεν υπήρχε η καταχώρηση στο TLB ή αν δεν υπήρχε καθόλου TLB (= δεν γίνεται), τότε θέλουμε 10 χρονικές μονάδες για ανάγνωση του PMT μέσω του MMU και άλλες 10 χρονικές μονάδες για να φέρουμε στη CPU τα δεδομένα. Συνολικά 20 χρονικές μονάδες.

|    |    |
|----|----|
| 0  | —  |
| 1  | —  |
| 2  | 10 |
| 3  | 11 |
| 4  | 7  |
| 5  | —  |
| 6  | —  |
| 7  | 2  |
| 8  | 12 |
| 9  | —  |
| 10 | 0  |

Αρχείο A

### ΑΡΧΕΙΑ

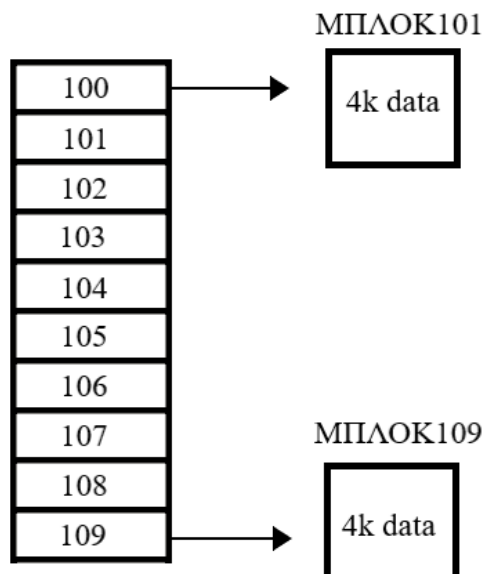
- Το αρχείο A ξεκινάει το πρώτο του κομμάτι από τη φυσική θέση 4
- Ακολουθούν τα κομμάτια στις θέσεις 7, 2, 10, 12, 6
- $5 * 4k = 20k$
- index-node(i-node): κόμβοι - δείκτες πληροφοριών για τα αρχεία

|   |                      |
|---|----------------------|
| 1 | image.txt<br>27/1/22 |
| 2 |                      |
| 3 | 8000                 |
| 4 | 25000                |
| 5 | 100000               |

### 1. Πληροφορίες αρχείου (όνομα, τύπος, ημ. δημιουργίας)

2. Άμεσοι δείκτες (10 ή 12): πρόκειται για δείκτες που δείχνουν σε μπλοκ, τα οποία περιέχουν δεδομένα.

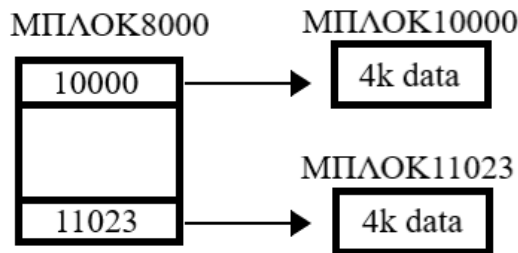
Για 10 δείκτες έχουμε τα εξής 10 μπλοκ:



- Τα “πρώτα” δεδομένα του αρχείου αποθηκεύονται με χρήση άμεσων δεικτών
- $10 \text{ μπλοκ} * 4k = 40k$

Αν ένα αρχείο είναι ως 40k (θεωρείται μικρό), η αποθήκευση του γίνεται με άμεσους δείκτες.

3. **Πρώτος έμμεσος δείκτης:** κάποια στιγμή επεκτείνεται το αρχείο και τα νέα δεδομένα δεν είναι απαραίτητο να αποθηκευτούν συνεχόμενα

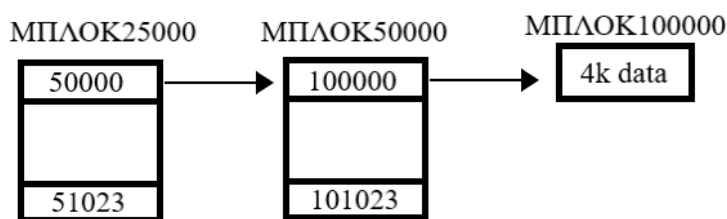


- Οι 1024 δείκτες δείχνουν σε μπλοκ δεδομένων

$$- 1024 \text{ μπλοκ} * 4k = 4M$$

Αν για ένα αρχείο απαιτείται μόνο 1 επίπεδο έμμεσης δεικτοδότησης, τότε το αρχείο έχει μέγεθος το πολύ  $40k + 4M$

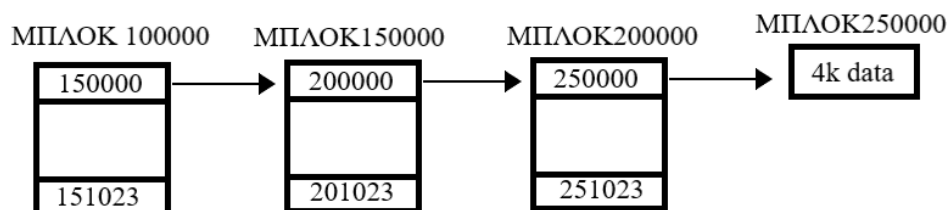
4. **Διπλός έμμεσος δείκτης:**



$$- 1024 * 1024 * 4 = 4GB$$

Αν για ένα αρχείο απαιτούνται μόνο 2 επίπεδα έμμεσης δεικτοδότησης, τότε το αρχείο έχει μέγεθος το πολύ  $40k + 4M + 4GB$

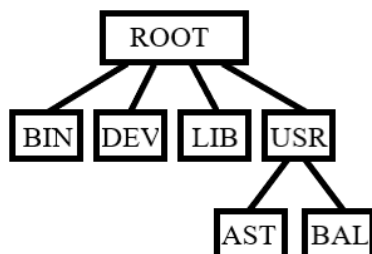
5. **Τριπλός έμμεσος δείκτης:**



$$- 1024 * 1024 * 1024 * 4 = 4.3T$$

Αν για ένα αρχείο απαιτούνται 3 επίπεδα έμμεσης δεικτοδότησης, τότε το αρχείο έχει μέγεθος το πολύ  $40k + 4M + 4GB + 4T$

#### Παράδειγμα



Να δώσετε μία δομή μπλοκ και i-nodes για τον εντοπισμό του αρχείου myfile.txt

Το αρχείο έχει μέσα του  $48K + 9M$  και η τιμή των άμεσων δεικτών είναι από 100 έως όσο χρειαστεί, ενώ η τιμή του έμμεσου δείκτη είναι 200. Να δείξετε το τρόπο αποθήκευσης του αρχείου και να βρείτε ποιος είναι ο τελευταίος δείκτης που αντιστοιχεί σε δεδομένα.

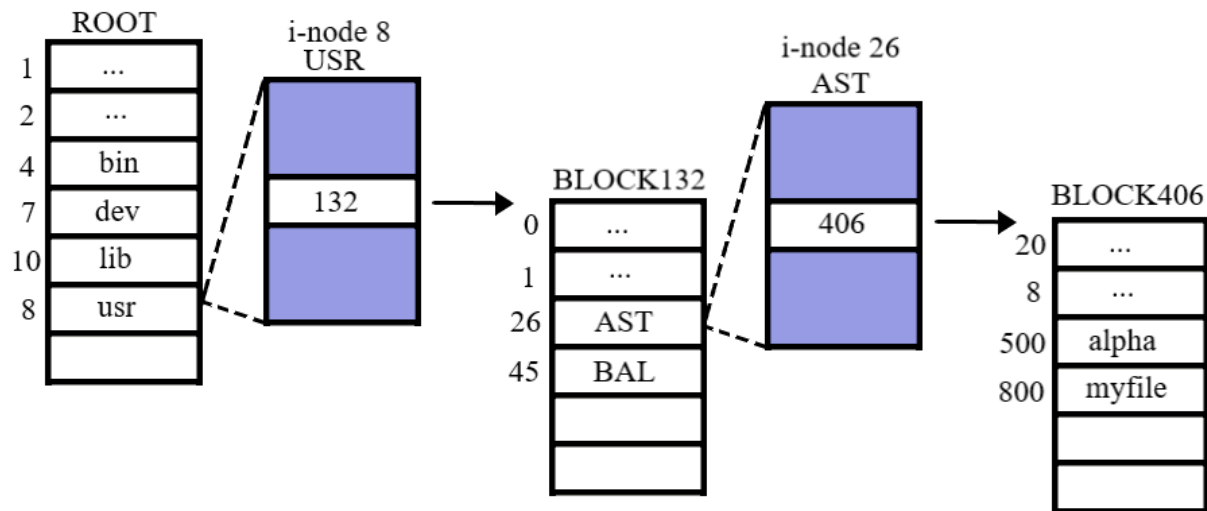
1)

Δεδομένα: μπλοκ δεικτών και καταλόγων

Μπλοκ: θα δείχνουν το περιεχόμενο του καταλόγου

Κατάλογοι: i-nodes θα δίνουν πληροφορίες για τον κατάλογο

i-nodes: πολλά γιατί θα περάσουν πολλούς καταλόγους

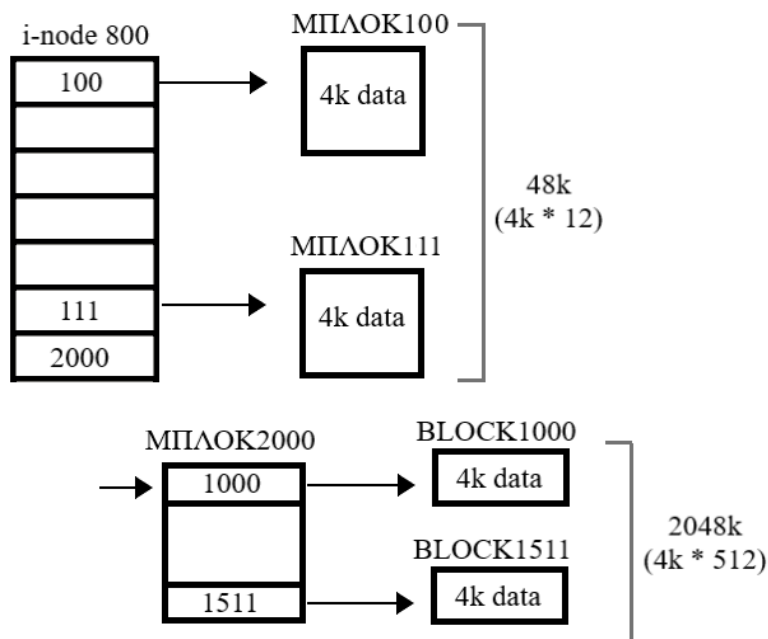


Ξεκινάμε από το block ROOT. Οι πληροφορίες για το φάκελο USR βρίσκονται στο i-node 8. Εκεί λέει ότι το USR είναι φάκελος, το μέγεθος του, ώρα δημιουργίας και η πληροφορία που μας ενδιαφέρει είναι ότι το block 132 διαθέτει τα περιεχόμενα του φακέλου USR.

Από το block 132: οι πληροφορίες για τον κατάλογο AST βρίσκονται στο i-node 26. Τελικά φτάνουμε στο block 406, που παρέχει τις πληροφορίες του φακέλου AST και εκεί διαβάζει ότι το i-node από το οποίο θα αναζητήσει τα δεδομένα του myfile.txt είναι το 800.

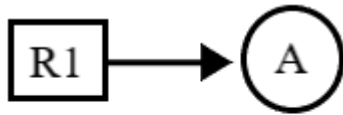
① Δεν είμαι σίγουρος για το παραπάνω διάγραμμα όσον αφορά το πως έχω ξεχωρίσει τα blocks και τα i-nodes

2)

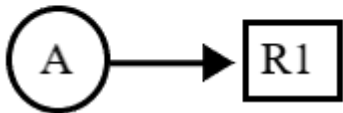


① Μία διεργασία περιμένει να ικανοποιηθεί μια συνθήκη για να προχωρήσει. Η συνθήκη αυτή όμως εξαρτάται από 1 γεγονός το οποίο δεν συμβαίνει ποτέ.

### ΠΙΘΑΝΑ ΑΔΙΕΞΟΔΑ



Η A δεσμεύει τον πόρο R1



Η A ζητάει να δεσμεύσει τον πόρο R1

- Για να κάνω χρήση ενός πόρου, κάνω down τον σηματοφορέα του
- Για να απελευθερώσω ένα πόρο, κάνω up τον σηματοφορέα του

| A           | B           |
|-------------|-------------|
| down(R1)    | down(R1)    |
| down(R2)    | down(R2)    |
| use(R1, R2) | use(R1, R2) |
| up(R2)      | up(R1)      |
| up(R1)      | up(R2)      |

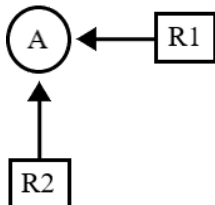
Έστω ξεκινάει η A και δεσμεύει τον πόρο R1 και τελειώνουν τα κβάντα του. Η B προσπαθώντας να δεσμεύσει τον R1, θα πάει για ύπνο. Η A και B διεκδικούν ίδιους πόρους, αλλά το σύστημα δεν μπαίνει σε αδιέξοδο.

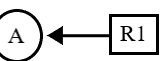
- Για να έχουμε αδιέξοδο, πρέπει να έχουμε τουλάχιστον 2 διεργασίες και 2 πόρους
- Σηματοφορείς: όλες οι διεργασίες πάνε για ύπνο → αδιέξοδος

| A           | B           |
|-------------|-------------|
| down(R1)    | down(R2)    |
| down(R2)    | down(R1)    |
| use(R1, R2) | use(R1, R2) |
| up(R2)      | up(R1)      |
| up(R1)      | up(R2)      |

- 1) Αν τα κβάντα αρκούν για να τρέξει ολόκληρη η A και μετά η B, έχουμε αδιέξοδο; Αναλύστε και κατασκευάστε το διάγραμμα.
- 2) Ίδιο με το 1), αλλά η A κόβεται μετά το down(R1)

- 1) Η A έχει δεσμεύσει πόρους που χρησιμοποιεί και τους ελευθερώνει



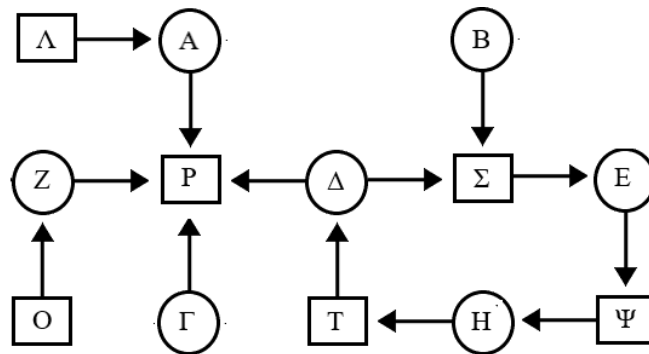
- 2) Η A δεσμεύει την R1 () και κόβεται. Η B δεσμεύει την R2, συνεχίζει κάνει down R1. Η αίτηση γίνεται. Δεν μπορεί να την δεσμεύσει και πάει για ύπνο. Η αίτηση έγινε (1).

Επανέρχεται η A, κάνει down R2. Η αίτηση γίνεται. Δεν μπορεί να την δεσμεύσει και πάει για ύπνο. Η αίτηση έγινε (2).

2 διεργασίες οι οποίες με διαφορετική σειρά αιτήσεων λειτούργησαν κανονικά και με άλλη οδήγησαν σε αδιέξοδο.

### ΠΑΡΑΔΕΙΓΜΑ

A κατέχει Λ, ζητάει P  
 B ζητάει Σ  
 Γ ζητάει P  
 Δ κατέχει T, ζητάει Σ, P  
 E κατέχει Σ, ζητάει Ψ  
 Z κατέχει O, ζητάει P  
 H κατέχει Ψ, ζητάει T



- 1) Ξεκινάμε από 1 διεργασία
- 2) Ελέγχουμε όλες τις διαδρομές που δημιουργούν εξερχόμενα τόξα. Αν σε κάποια διαδρομή ένας κόμβος βρεθεί 2 φορές, πιθανότατα έχουμε αδιέξοδο.

Γ: L{P}

Από P δεν υπάρχει εξερχόμενο τόξο άρα η Γ δεν συμμετέχει σε αδιέξοδο.

H: L{T, Δ}

Δ: L{P, Σ}

- 1) P: L = {T, Δ, P} → όχι αδιέξοδος
- 2) Σ: L = {T, Δ, Σ, E, Ψ, H}

- Ξεκινήσαμε από H → H (κύκλος)

| H         | Δ         | E         |
|-----------|-----------|-----------|
| down(Ψ)   | down(T)   | down(Σ)   |
| down(T)   | down(Σ)   | down(Ψ)   |
| use(Ψ, T) | use(T, Σ) | use(Σ, Ψ) |
| up(Ψ)     | up(T)     | up(Σ)     |
| up(T)     | up(Σ)     | up(Ψ)     |

Αν η H δεσμεύσει την Ψ και κοπεί, τρέξει η Δ δεσμεύσει T και κοπεί, τρέξει η E δεσμεύσει Σ και κοπεί, ΑΔΙΕΞΟΔΟΣ,