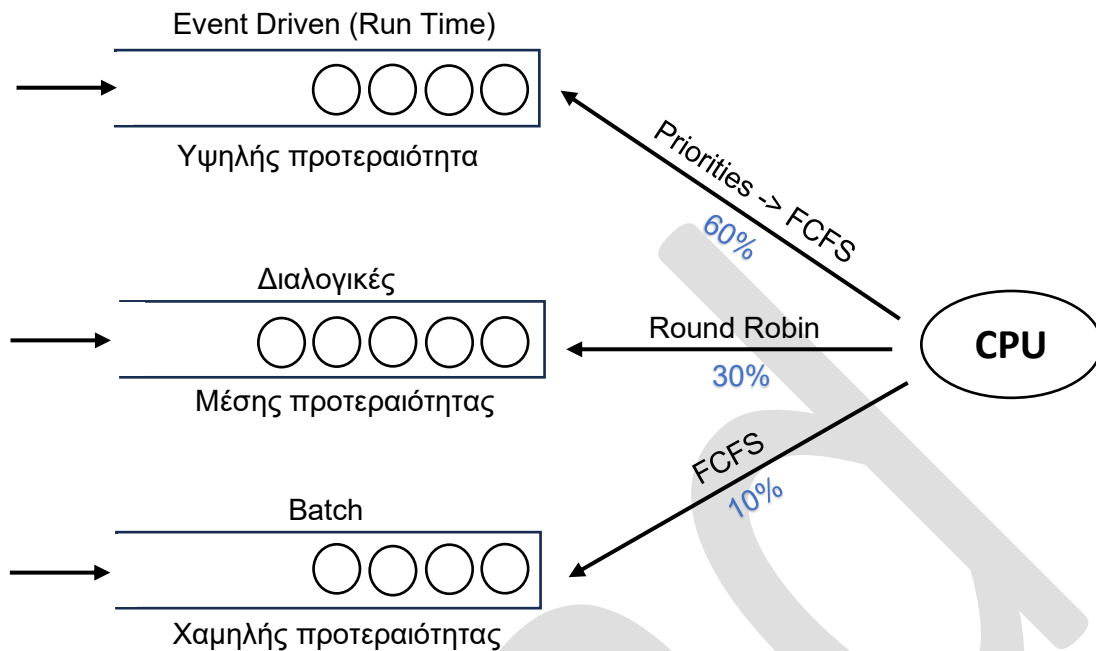


Λειτουργικά Συστήματα – Τμήμα Β' (Μαξ-Ω)

Διάλεξη 4

Ουρές με διεργασίες

Μπορούμε στην κατάσταση READY να έχουμε ουρές με διεργασίες, ώστε να μην βρίσκονται «χύμα».



Παράδειγμα

Έστω ότι έχουμε 100 κβάντα.

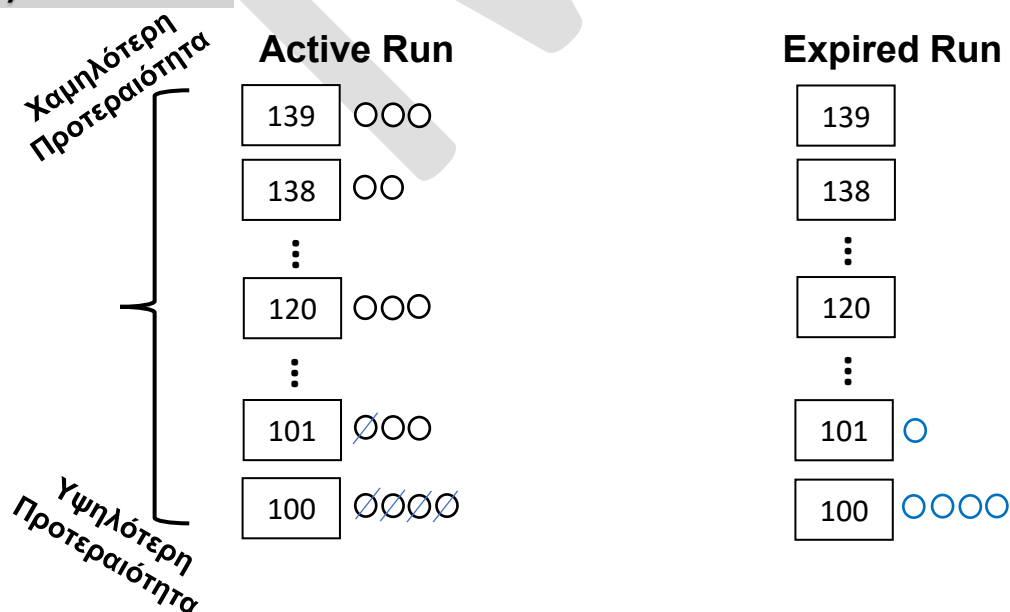
- **Batch:** 10 κβάντα η 1^η διεργασία

- **Διαλογικές:** 30 κβάντα, 1-1 κβάντο σύμφωνα με τον αλγόριθμο Round Robin (αν για παράδειγμα οι διεργασίες είναι λιγότερες από 30 υπάρχει περίπτωση κάποιες αντί για 1 να παίρνουν και 2 κβάντα)

- **Event driven:** όλα τα κβάντα η 1^η διεργασία

* Τα ποσοστά δεν είναι πάντα σταθερά. Αν για παράδειγμα κάποια ουρά είναι άδεια μοιράζονται αντίστοιχα και τα ποσοστά. Π.χ. τελειώνουν οι διαλογικές --> 3 προς 1 στα Batch

O(1) scheduler



Κάθε διεργασία που παίρνει ένα κβάντο φεύγει από την Active Run ουρά και πηγαίνει στην αντίστοιχη Expired. Κάποια στιγμή θα τελειώσουν όλες οι διεργασίες από την Active πηγαίνοντας στην Expired. Τότε το ΛΣ «σβήνει» την Active Run ουρά κάνοντας την Expired, και αντίστοιχα την Expired την κάνει Active. Αυτή η εναλλαγή θα γίνεται κάθε φορά που τελειώνουν οι διεργασίες της Active ουράς.

Quantum ανάλογα με το priority

if priority < 120

$$q = (140 - \text{priority}) * 20 \text{ ms}$$

else

$$q = (140 - \text{priority}) * 5 \text{ ms}$$

Παράδειγμα

$$105: q = (140 - 105) * 20 = 35 * 20 = 700 \text{ ms}$$

$$130: q = (140 - 130) * 5 = 10 * 5 = 50 \text{ ms}$$

$$120: q = (140 - 120) * 5 = 20 * 5 = 100 \text{ ms}$$

$$119: q = (140 - 119) * 20 = 21 * 20 = 420 \text{ ms}$$

Παράδειγμα – Άσκηση

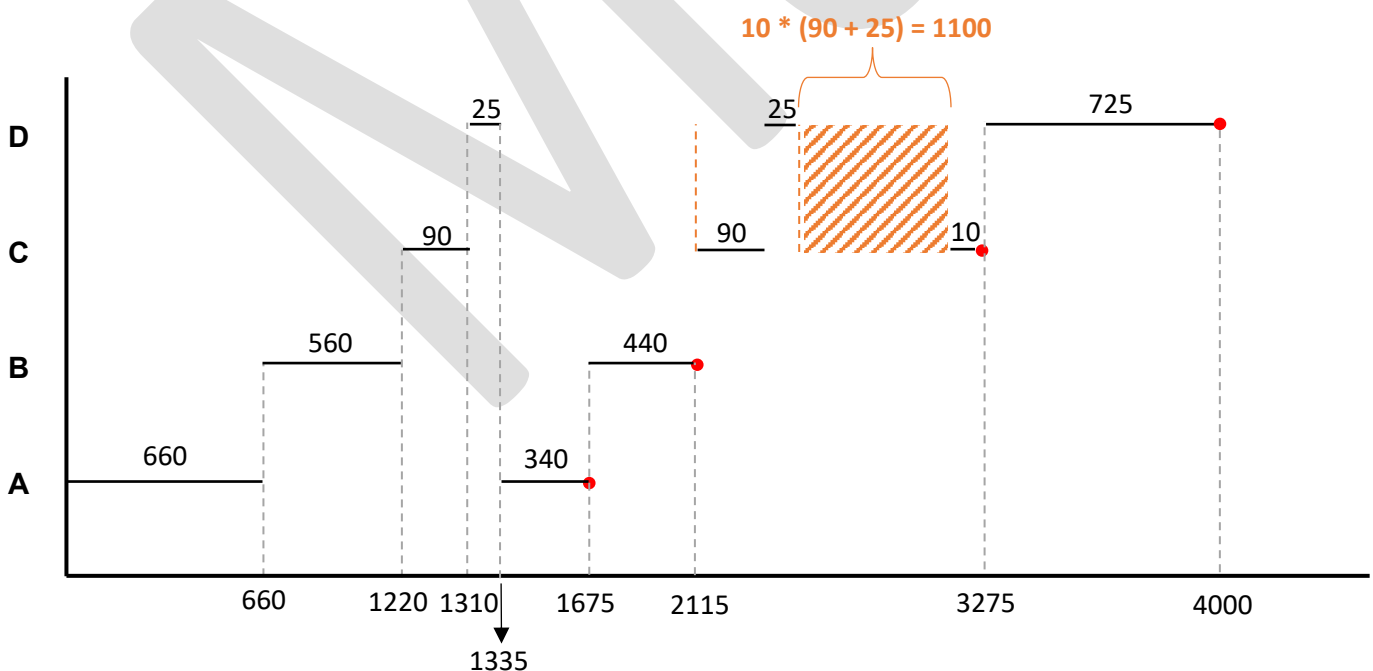
Έχουμε διεργασίες A, B, C, D με αριθμούς προτεραιοτήτων 107, 112, 122 και 135 αντίστοιχα. Όλες έχουν χρόνο εκτέλεσης 1000 ms.

$$q_A = (140 - 107) * 20 = 600 \text{ ms}$$

$$q_B = (140 - 112) * 20 = 560 \text{ ms}$$

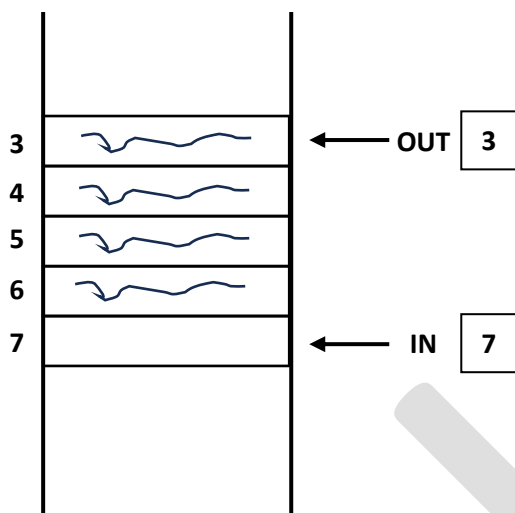
$$q_C = (140 - 122) * 5 = 90 \text{ ms}$$


$$q_D = (140 - 135) * 5 = 25 \text{ ms}$$



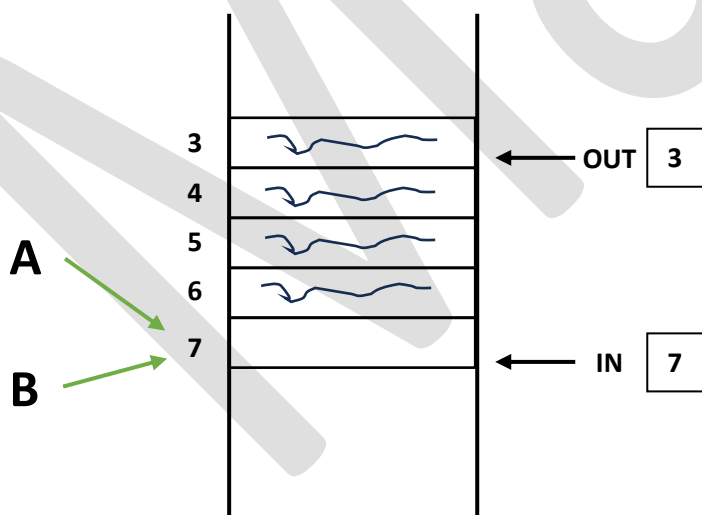
* Με κόκκινη βουλίτσα σημειώνεται η χρονική στιγμή που τελειώνει η κάθε διεργασία

Διεργασιακή Επικοινωνία



Ας υποθέσουμε ότι έχουμε ένα πίνακα της παραπάνω μορφής στον οποίο αποθηκεύουμε το όνομα του αρχείου μας. Κάθε διεργασία που θέλει να τυπώσει βάζει ένα αρχείο σε ένα ειδικό κατάλογο και λέει στο ΛΣ να το εκτυπώσει. Έστω όπου  υπάρχει το όνομα κάποιου αρχείου (άρα η θέση 7 είναι κενή). Το ΛΣ όταν τυπώσει τα αρχεία θα τα «σβήσει» μετά από μόνο του οπότε οι διεργασίες δεν θα ξανά ασχοληθούν με αυτά. Το ΛΣ για να γνωρίζει ότι τυπώνει το αρχείο στη θέση 3, αρκεί να έχουμε μια μεταβλητή OUT που να έχει την τιμή 3. Αντίστοιχα, θα πρέπει να έχουμε και μια μεταβλητή IN που θα μας λέει μια καινούρια εκτύπωση σε ποια θέση πρέπει να πάει (η IN δείχνει στη θέση 7, δηλαδή στην επόμενη κενή θέση). Επομένως, μια διεργασία θέλει να τυπώσει θα διαβάσει την τιμή της μεταβλητής IN, να πάει σε αυτή τη θέση να βάλει το όνομα του αρχείου και να αυξήσει την τιμή της μεταβλητής κατά 1.

Ας υποθέσουμε ότι είναι έτοιμες 2 διεργασίες οι A και B.

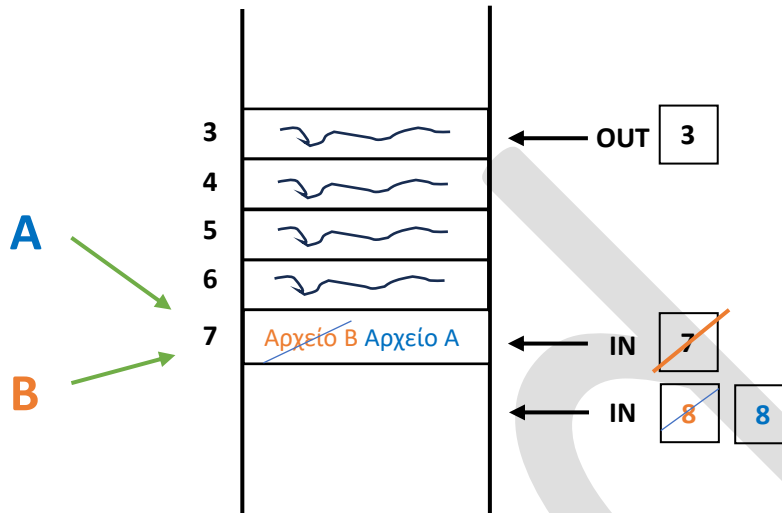


Στο RUN θα είναι μία από τις δύο. Αυτή που είναι πρώτη στο RUN και θέλει να τυπώσει, αυτή θα είναι και η πρώτη που θα φτάσει στη μεταβλητή, και μετά θα έρθει η άλλη.

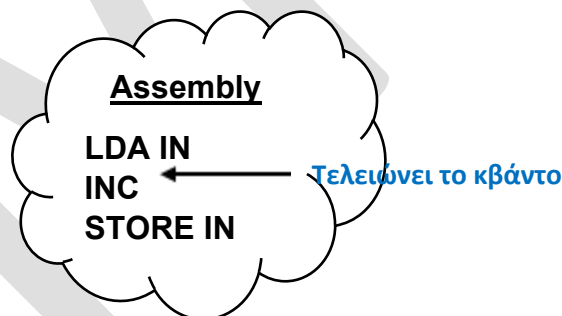
Κάθε διεργασία παίρνει 1 κβάντο. Το κρίσιμο σημείο είναι τι γίνεται όταν τελειώνει το κβάντο μιας διεργασίας. Μια διεργασία μπορεί να διακοπεί οποιαδήποτε χρονική στιγμή.

Παράδειγμα

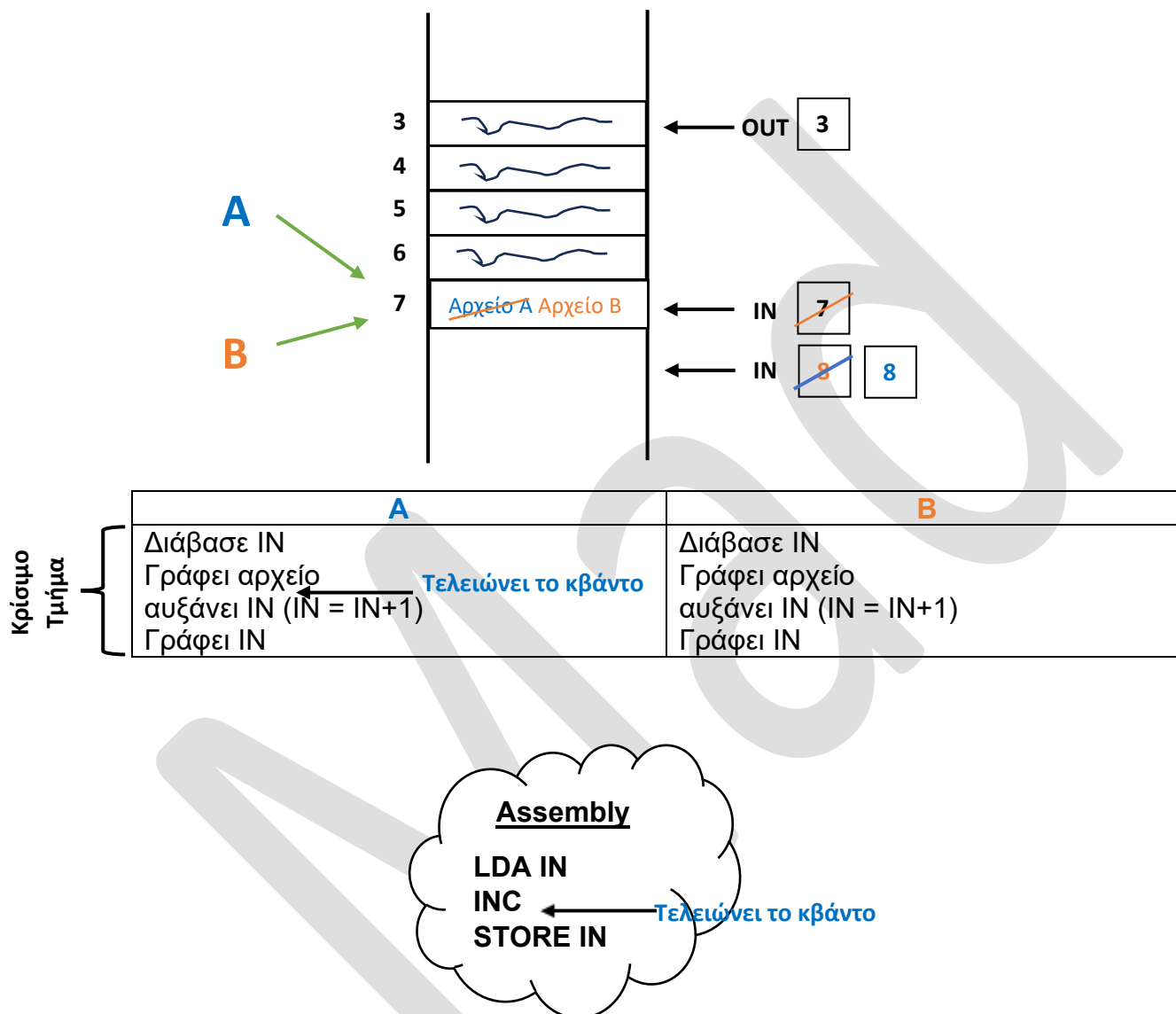
Έρχεται η διεργασία A, διαβάζει την μεταβλητή IN και «βρίσκει» 7. Εκείνη τη στιγμή τελειώνει το κβάντο της. Έρχεται η διεργασία B, διαβάζει την μεταβλητή IN και «βρίσκει» 7. Πάει και βάζει το αρχείο της και κάνει το IN 8. Συνεχίζει τη δουλειά της, μέχρι να τελειώσει το κβάντο της. Αφού έρθει ξανά η σειρά της A, συνεχίζει από εκεί που έμεινε. Πάει δηλαδή στη θέση 7, που είχε διαβάσει πριν τελειώσει το κβάντο και γράφει το αρχείο της. Έπειτα, κάνει το IN 8 και συνεχίζει τη δουλειά της. Παρατηρούμε ότι το αρχείο της B δε θα τυπωθεί ποτέ.



		A	B
Κρίσιμο Τμήμα	Διάβασε IN	← Τελειώνει το κβάντο	Διάβασε IN
	Γράφει αρχείο		Γράφει αρχείο
	αυξάνει IN ($IN = IN + 1$)		αυξάνει IN ($IN = IN + 1$)
	Γράφει IN		Γράφει IN



Ας δούμε και την περίπτωση που το κβάντο τελειώνει μόλις γράψει η διεργασία A το όνομα του αρχείου της. Η διεργασία A διαβάζει το IN (έχει την τιμή 7) και γράφει το αρχείο της στην αντίστοιχη θέση. Σε αυτό το σημείο τελειώνει το κβάντο και «κόβεται». Έρχεται η διεργασία B, διαβάζει το IN (έχει την τιμή 7) και γράφει το αρχείο της στην αντίστοιχη θέση. Έπειτα αυξάνει το IN κατά 1 (έχει δηλαδή πλέον την τιμή 8) και συνεχίζει μέχρι να τελειώσει το κβάντο της. Αφού ξαναέρθει η σειρά της διεργασίας A, συνεχίζει από εκεί που έμεινε. Έχοντας διαβάσει το IN (με τιμή 7) το αυξάνει κατά 1 (έχει δηλαδή πάλι την τιμή 8) και συνεχίζει τη δουλειά της. Παρατηρούμε ότι δε θα τυπωθεί ποτέ το αρχείο της A.



Κρίσιμο τμήμα ονομάζουμε το κομμάτι εκτέλεσης μιας διεργασίας κατά την εκτέλεση του οποίου η διεργασία θα είναι καλό να είναι μόνη της, να μη μπει δηλαδή κάποια άλλη εργασία στο αντίστοιχο δικό της κρίσιμο τμήμα. Αυτό ονομάζεται **αμοιβαίος αποκλεισμός** (η μία αποκλείει την άλλη).

Χαρακτηριστικά καλής λύσης

1. Αμοιβαίος αποκλεισμός
2. Καμία υπόθεση για ταχύτητα ή πλήθος CPU
3. Διεργασία σε μη κρίσιμο τμήμα δεν αναστέλλει άλλες
4. Είσοδος στο κρίσιμο τμήμα σε πεπερασμένο χρόνο
5. Crash recovery (σε μη κρίσιμο σημείο) ~ παρόμοιο με το 3 ~

→ Αν crashάει, οι υπόλοιπες διεργασίες να τρέχουν κανονικά

Λύσεις (1st part)

1. Απενεργοποίηση διακοπών

Μια διεργασία θα μπορούσε όταν μπαίνει στο κρίσιμο τμήμα της να απενεργοποιεί τις διακοπές και όταν βγαίνει να τις ξανά ενεργοποιεί

Πρόβλημα: αν είναι στο Κ.Τ. και απενεργοποιήσει τις διακοπές και εκείνη την ώρα τελειώσει το κβάντο της δεν θα το καταλάβει κανένας, συνεπώς δε θα μπορέσει να τρέξει και καμία άλλη διεργασία. Μόνο όταν βγει από το Κ.Τ. μπορεί να τις ξανά ενεργοποιήσει (Δεν καλύπτει

όλα τα χαρακτηριστικά καλής λύσης)

2. Μεταβλητές κλειδώματος

Μπορώ να έχω κάποια μεταβλητή στη μνήμη που να υποδηλώνει ότι κάποιος είναι μέσα στο Κ.Τ. του. Μπαίνοντας στο Κ.Τ. μια διεργασία κάνει την μεταβλητή 0 και βγαίνοντας 1. Το πρόβλημα παραμένει παρόμοιο με του 1.

3. Αυστηρή Εναλλαγή

Αφορά μόνο 2 διεργασίες και δε λειτουργεί με παραπάνω. Παρατηρούμε infinite loops, όπου οι εντολές θα τρέχουν συνέχεια. Ο λόγος έχουμε infinite loops εδώ πέρα (σε αντίθεση με τα προγράμματα που ήδη γνωρίζουμε) είναι για να ελέγξουμε όλους τους δυνατούς συνδυασμούς, δηλαδή μήπως τύχει και διακοπεί σε κάποιο σημείο του και τρέξει ο βρόγχος της άλλης διεργασίας ο οποίος μπορεί και αυτός να διακοπεί σε οποιοδήποτε σημείο του και να ξανατρέξει η προηγούμενη.

Η πρώτη ελέγχει να δει αν το turn είναι διαφορετικό από το 0 (το turn αποτελεί μια κοινή μεταβλητή που χρησιμοποιείται και από τις δύο διεργασίες. Θα έχει την τιμή 0 όταν μπορεί η μια διεργασία να μπει μέσα και 1 όταν θα μπορεί η άλλη). Η 1^η διεργασία που ας πούμε έχει το όνομα 0, αν βρει το turn διαφορετικό του 0 τότε περιμένει (το wait μπορεί να είναι κάποια ρουτίνα του ΛΣ το οποίο βάζει τη διεργασία σε μια άλλη κατάσταση αναμονής μέχρι η μεταβλητή turn να αλλάξει τιμή, δηλαδή στη περίπτωση μας μέχρι να γίνει 0). Αν το turn έχει την τιμή 0, τότε περνάει η διεργασία στο Κ.Τ. της. Βγαίνοντας από το Κ.Τ. της το turn κάνει 1 και συνεχίζει στο μη Κ.Τ. κάνοντας άλλες «δουλειές» (π.χ. υπολογισμούς). Αυτό που στοχεύουμε να πετύχουμε είναι όταν η 1^η διεργασία είναι στο Κ.Τ. της να μη μπορεί να μπει και η 2^η στο αντίστοιχο δικό της. Αν εξετάσουμε και τις εντολές της 2^{ης} διεργασίας θα δούμε ότι είναι παρόμοιες με αυτές της 1^{ης} και απλά αλλάζουν οι τιμές της μεταβλητής turn.

Πρόβλημα: έρχεται η 2^η διεργασία μπαίνει στο Κ.Τ. της, βγαίνει, κάνει το turn 0, μπαίνει στο μη Κ.Τ. της. Γίνεται ο έλεγχος και επειδή το turn είναι διάφορο του 1 μπαίνει σε κατάσταση αναμονής. Μπορεί να ξαναμπει στο Κ.Τ. της; **ΟΧΙ**. Πρέπει να μπει πρώτα και να βγει η 1^η διεργασία κάνοντας το turn 1. (δεν είναι καλή γιατί μπορεί να παραβιάσει το 3, 5)

```
while (TRUE) {  
    while (turn<>0) wait  
    critical_section  
    turn=1  
    non_critical_section  
}  
while (TRUE) {  
    while (turn<>1) wait  
    critical_section  
    turn=0  
    non_critical_section  
}
```

Εντολές 1^{ης} διεργασίας

Εντολές 2^{ης} διεργασίας

4. Λύση του Peterson

Λειτουργεί και με παραπάνω από δύο διεργασίες (ο παρακάτω ψευδοκώδικας είναι σχεδιασμένος **για 2 μόνο**, αλλά με κάποιες αλλαγές είναι εφικτό και για περισσότερες. Στο μάθημα μας αφορά μόνο η περίπτωση με δύο διεργασίες για τη λύση του Peterson). Επίσης, δεν προϋποθέτει αυστηρή εναλλαγή, δηλαδή αν μια εργασία «βιάζεται» να μπει στο Κ.Τ. πολλές φορές, μπορεί να μπει όσες φορές θέλει.

```
int turn, boolean interested [N]
enter (int process) {
    other=1-process
    interested[process]=TRUE
    turn=process
    while (turn=process and
interested[other]=TRUE) {}
}

leave (int process) {
    interested[process]=FALSE
}
```

Η κάθε διεργασία έχει έναν αριθμό, το process είναι αυτός όπως φαίνεται παραπάνω. Όπως παρατηρούμε η λύση αυτή αποτελείται από δύο ρουτίνες, την enter και την leave. Μια διεργασία που θέλει να μπει στο Κ.Τ. της καλεί την ρουτίνα enter και όταν βγαίνει καλεί την leave. Ο αριθμός της διεργασίας (process) είναι παράμετρος της ρουτίνας enter (π.χ. αν την καλέσει η διεργασία 0, η παράμετρος θα έχει την τιμή 0 άρα όπου process θα θεωρήσουμε 0).

Έχουμε μια κοινή μεταβλητή turn, που δείχνει ποιανού σειρά είναι να μπει (δεν χρησιμοποιείται όμως για αυστηρή εναλλαγή, δηλαδή η ίδια διεργασία μπορεί να μπει π.χ. 10 φορές). Υπάρχει και ένας πίνακας τύπου boolean (True ή False) με μέγεθος N (N = αριθμός διεργασιών)

Έστω έρχεται η διεργασία 0 και θέλει να μπει στο Κ.Τ. της. Πρέπει να καλέσει την ρουτίνα enter με παράμετρο 0. Το other, που αποτελεί τοπική μεταβλητή, θα πάρει την τιμή 1 (αφού $1 - process = 0$) και δείχνει ουσιαστικά την άλλη διεργασία. Το interested[0] το κάνω TRUE, που τυπικά σημαίνει πως ενδιαφέρομαι να μπω στο Κ.Τ. της διεργασίας. Έπειτα κάνω το $turn = process$ (δηλαδή ίσο με 0) για να μπω «με το ζόρι». Στη συνέχεια έχουμε ένα ατέρμων βρόγχο while, που όσο η συνθήκη είναι ΑΛΗΘΗΣ τρέχει χωρίς να κάνει τίποτα και δε μπορούμε να βγούμε από το loop. Όπως βλέπουμε, για να είναι η συνθήκη ΑΛΗΘΗΣ πρέπει και το $turn = process$ να είναι ΑΛΗΘΗΣ και το interested[other] να είναι ΑΛΗΘΗΣ. Παρατηρούμε ότι αν το $turn = process$ είναι TRUE (δηλαδή ενδιαφέρομαι να μπω στο Κ.Τ.) όμως και ότι το interested[other] == TRUE (δηλαδή ενδιαφέρεται και η άλλη διεργασία να μπει στο Κ.Τ. της) μπαίνω στο while loop. Πώς όμως θα βγει από την συνθήκη; Όσο τρέχει η διεργασία μέσα στο loop, μπορεί να τελειώσει το κβάντο της. Όταν έρθει η σειρά της άλλης διεργασίας, μπορεί να αλλάξει το turn, όπως και την τιμή της θέσης που έχει στον πίνακα interested[]. Όταν η συνθήκη του while loop είναι ΨΕΥΔΗΣ, η διεργασία μπαίνει στο Κ.Τ. της.

Η ρουτίνα leave το μόνο που κάνει είναι βγαίνοντας η διεργασία από το Κ.Τ. της να κάνει το interested[process] (δηλαδή της ίδιας της διεργασίας) ίσο με FALSE, δηλαδή ότι πλέον δεν ενδιαφέρεται να μπει στο Κ.Τ. της.

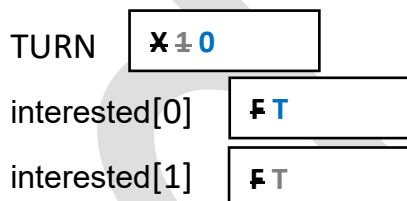
Υπό κανονικές συνθήκες, μπαίνοντας μια διεργασία στο Κ.Τ. της η άλλη δε μπορεί να μπει. *Τι γίνεται αν τελειώσει το κβάντο μιας διεργασίας όμως και «κοπεί»;* Θα πρέπει να έχουμε υπόψιν μας ότι κοινή μεταβλητή είναι μόνο το turn. Τον πίνακα interested[N], ενώ ο καθένας μπορεί να δει την τιμή του άλλου, μπορεί να αλλάξει μόνο την δικιά του.

1^η περίπτωση που μας αφορά (να «κοπεί» πριν το turn=process)

```
int turn, boolean interested [N]
```

```
enter (int process) {  
    other=1-process  
    *interested[process]=TRUE  
    turn=process  
    while (turn=process and  
interested[other]=TRUE) {*2  
        }*3  
}
```

```
leave (int process) {  
    interested[process]=FALSE  
}
```

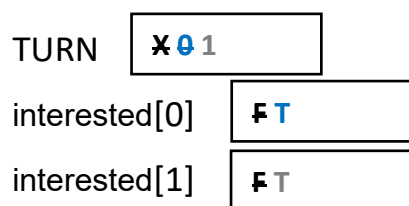


Το interested[0] και interested[1] έχουν αρχικές τιμές FALSE. Η τιμή του turn αρχικά δε μας αφορά. Ξεκινάει έστω η 0 να θέλει να μπει στο κρίσιμο τμήμα της και καλεί την ρουτίνα enter με παράμετρο 0. Το other (θα γίνει 1) είναι τοπική μεταβλητή, όποτε δε μας ενδιαφέρει αν «κοπεί» η διεργασία κάπου εκεί καθώς δε μας κάνει καμία διαφορά. Μετά το interested[0] γίνεται TRUE. Υποθέτουμε ότι «κόβεται» σε αυτό το σημείο*. Θα τρέξουν τώρα κάποιες άλλες υποθετικές διεργασίες και θα έρθει η σειρά της 1, που θέλει να μπει στο Κ.Τ. της. Καλεί την enter με παράμετρο το 1, το other θα γίνει 0, το interested[1] θα γίνει TRUE. Το turn μετά γίνεται ίσο με το process (δηλαδή 1). Φτάνουμε στο while loop όπου βλέπουμε πως η συνθήκη είναι ΑΛΗΘΗΣ (turn ισούται με process, interested[0] είναι TRUE) οπότε μπαίνει και παραμένει στο loop μέχρι να τελειώσει το κβάντο της^{*2}. Ξανά γυρνάμε στην 0, η οποία συνεχίζει από το σημείο που κόπηκε. Το turn το κάνει 0. Πάει στο while loop, όπου η συνθήκη είναι πάλι ΑΛΗΘΗΣ (turn ισούται με process, interested[1] είναι TRUE) και παραμένει εκεί μέχρι να τελειώσει το κβάντο της^{*3}. Έρχεται πάλι η διεργασία 1, που είχε «κοπεί» μέσα στο loop. Η συνθήκη όμως τώρα είναι ΨΕΥΔΗΣ, αφού το turn δεν ισούται με το process (turn = 0, process = 1), συνεπώς μπαίνει στο Κ.Τ. της. (Δηλαδή 0 → 1 → 0 → 1 και τελικά μπήκε στο Κ.Τ. της)

2^η περίπτωση που μας αφορά (να «κοπεί» μετά το turn=process)

```
int turn, boolean interested [N]
enter (int process) {
    other=1-process
    interested[process]=TRUE
    *turn=process
    *while (turn=process and
interested[other]=TRUE) {*2}
}

leave (int process) {
    interested[process]=FALSE
}
```

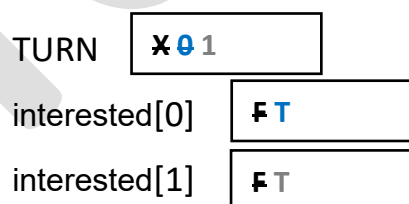


Το interested[0] και interested[1] έχουν αρχικές τιμές FALSE. Η τιμή του turn αρχικά δε μας αφορά. Ξεκινάει έστω η 0 να θέλει να μπει στο κρίσιμο τμήμα της και καλεί την ρουτίνα enter με παράμετρο 0. Το other θα γίνει 1, το interested[0] θα γίνει **TRUE** και το turn θα γίνει **0**. Σε αυτό το σημείο, υποθέτουμε ότι τελειώνει το κβάντο και «κόβεται» η διεργασία 0. Έρχεται η διεργασία 1, καλεί την enter με παράμετρο 1. Το other γίνεται 0, το interested[1] γίνεται **TRUE** και το turn γίνεται ίσο με **1**. Στο while loop η συνθήκη είναι ΑΛΗΘΗΣ (turn ισούται με process = 1 και interested[0] είναι TRUE), οπότε παραμένει μέσα σε αυτό ώσπου κάποια στιγμή τελειώνει το κβάντο της ^{*2}. Θα τρέξουν τυχών άλλες διεργασίες και θα έρθει πάλι η σειρά της 0, που θα συνεχίσει από το σημείο που κόπηκε. Φτάνει δηλαδή κατευθείαν στο while loop όπου η συνθήκη είναι ΨΕΥΔΗΣ, αφού το turn(1) δεν ισούται με το process(0), και μπαίνει έτσι στο Κ.Τ. της. (Δηλαδή 0 → 1 → 0 και τελικά μπήκε στο Κ.Τ. της)

3^η περίπτωση που μας αφορά (να «κοπεί» μέσα στη συνθήκη το while loop)

```
int turn, boolean interested [N]
enter (int process) {
    other=1-process
    interested[process]=TRUE
    turn=process
    *while (turn=process and
interested[other]=TRUE) {*2}
}

leave (int process) {
    interested[process]=FALSE
}
```



Το interested[0] και interested[1] έχουν αρχικές τιμές FALSE. Η τιμή του turn αρχικά δε μας αφορά. Ξεκινάει έστω η 0 να θέλει να μπει στο κρίσιμο τμήμα της και καλεί την ρουτίνα enter με παράμετρο 0. Το other θα γίνει 1, το interested[0] θα γίνει **TRUE** και το turn θα γίνει **0**. Μπαίνοντας στην συνθήκη του while loop μπορεί να «κοπεί» και στον έλεγχο (καθώς ο ψευδοκώδικας παραπάνω δεν αποτελεί γλώσσα μηχανής, στην οποία το while loop αποτελείται από περισσότερες εντολές). Έστω ότι διαβάζει τα turn και process και τα συγκρίνει βλέποντας ότι είναι ίσα, άρα TRUE, και «κόβεται»^{*}. Θα έρθει μετά η σειρά της 1, που θα καλέσει την ρουτίνα enter με παράμετρο 1, το other θα γίνει 0, το interested[1] θα

γίνει **TRUE** και το turn θα γίνει **1**. Βλέπουμε πως η συνθήκη του while loop θα είναι ΑΛΗΘΗΣ (turn ίσο με process, interested[0] = TRUE) άρα θα παραμείνει μέσα σε αυτό μέχρι να τελειώσει το κβάντο*2. Ξαναγυρνάμε στη 0, που συνεχίζει από εκεί που είχε κοπεί. Είχε βρει ότι turn ισούται με process, βρίσκει τώρα ότι το interested[1] είναι TRUE, άρα η συνθήκη της while loop είναι ΑΛΗΘΗΣ και τρέχει, μετά ξαναελέγχει αλλά αυτή τη φορά βλέπει ότι το turn δεν είναι ίσο με το process (αφού είχε μπει προηγουμένως η διεργασία 1 στη ρουτίνα enter κάνοντας το 1), άρα μπαίνει στο Κ.Τ. της.

Βλέποντας τις παραπάνω περιπτώσεις παρατηρούμε πως δεν γίνεται να μπουν ταυτόχρονα στο Κ.Τ. τους οι δυο διεργασίες.

Χρήσιμα Links:

https://youtu.be/K6u1A5_5zV8?si=uUbs9-bdNBSUhelk

<https://www.geeksforgeeks.org/g-fact-70/?ref=lbp>

<https://www.geeksforgeeks.org/petersons-algorithm-in-process-synchronization/?ref=lbp>