Λειτουργικά Συστήματα – Τμήμα Β΄ (Μαξ-Ω)
Διάλεξη 5

Λύσεις (2nd part – συνέχεια διάλεξης 4)

5. TSL εντολή

Όταν ένα πρόβλημα software δεν είναι εύκολο να λυθεί, έρχεται σε βοήθεια το hardware. Έτσι, δημιούργησαν οι σχεδιαστές CPU μια εντολή με όνομα TSL (Test and Set Lock), δηλαδή «Έλεγχος και Κλείδωμα». Γενικώς το set σημαίνει στο hardware κυρίως ότι κάτι έχει την τιμή 1 (π.χ. κάνω μια μεταβλητή set, την κάνω 1).

Σε assembly:

```
enter
    tsl register, flag
    cmp register, #0
    jnz enter
    ret

leave
    move flag, #0
    ret
```

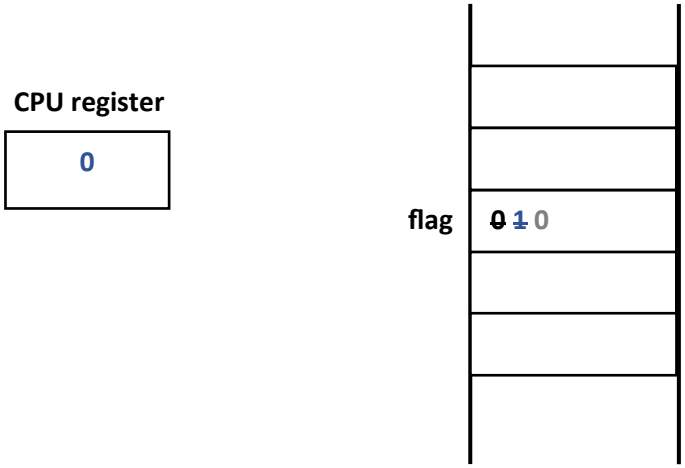
Όπως και τη λύση του Peterson, έχουμε δύο ρουτίνες την enter και την leave. Όταν μια διεργασία θέλει να μπει στο Κ.Τ. της θα καλεί την enter και βγαίνοντας από αυτό θα καλεί την leave. Έχουμε μια θέση της μνήμης που ονομάζεται flag. Αυτή η flag θα είναι η μεταβλητή κλειδώματος, όταν είναι 0 τότε μια διεργασία θα μπορεί να μπει στο Κ.Τ. της. Η εντολή TSL κάνεις δύο δουλειές. Αρχικά, παίρνει το περιεχόμενο της μεταβλητής flag και να το μεταφέρει στον καταχωρητή register και έπειτα τη θέση μνήμης flag την κάνει 1 (όποια τιμή και αν έχει, θα γίνει 1). Η εντολή cmp (compare) συγκρίνει δύο πράγματα μεταξύ τους, στην περίπτωση μας έναν καταχωρητή και την πραγματική τιμή 0. Η εντολή jnz enter (jump not zero) αυτό που κάνει είναι να βλέπει αν δεν είναι μηδέν (η προηγούμενη σύγκριση) τότε θα πάμε ξανά στην αρχή της ρουτίνας enter (δηλαδή στην εντολή tsl). Αν ο καταχωρητής έχει την τιμή 0, το jnz δε θα κάνει τίποτα και θα πάει στην επόμενη εντολή την ret (return) που χρησιμοποιείται για την επιστροφή από υπορουτίνες. Άρα αν περάσω από το return, θα επιστρέψω εκεί που ήμουν και θα μπορέσει η διεργασία να μπει στο Κ.Τ. της. Βγαίνοντας από το Κ.Τ. της καλεί την leave η οποία εκτελεί μια εντολή που λέγεται move και μεταφέρει την τιμή 0 στο flag και έπειτα επιστρέφει (ret).

Ουσιαστικά έχουμε:

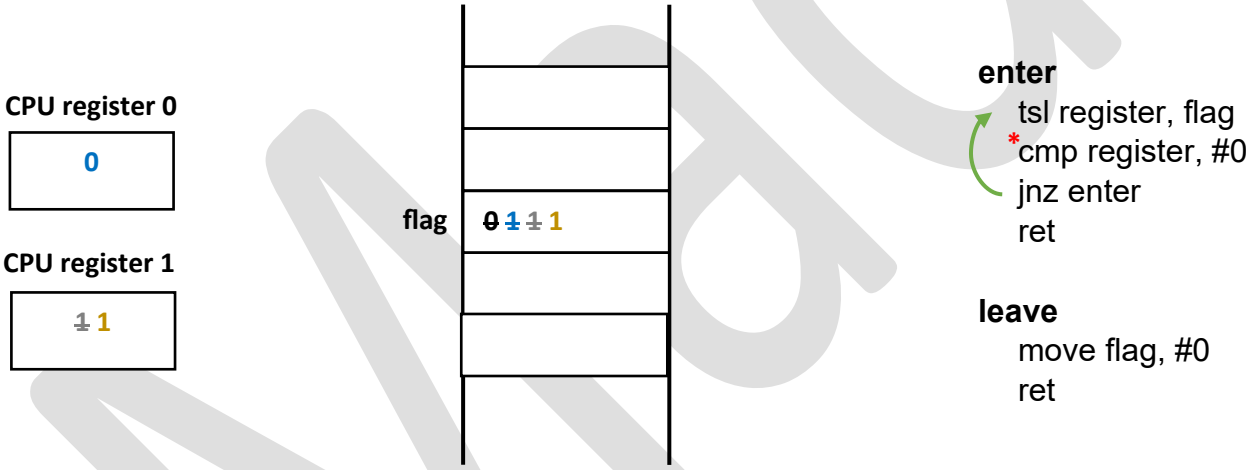
```
enter()
critical_section
leave()
```

**** Η διεργασία που τρέχει, μπορεί να «κοπεί» είτε πριν είτε μετά την εντολή tsl register, flag και όχι ενδιάμεσα όπως στο while loop που είδαμε στη λύση του Peterson.***

***** επειδή τα ονόματα των εντολών της γλώσσας μηχανής είναι copyrighted, TSL ονομάζεται η εντολή σε κάποιον συγκεκριμένο CPU, ενώ σε κάποιον άλλον μπορεί να έχει άλλο όνομα***



Ας ξεκινήσουμε με την απλή περίπτωση που μια διεργασία θέλει οπωσδήποτε να μπει στο Κ.Τ. της. Καμία διεργασία δεν είναι στο Κ.Τ. της, άρα το flag είναι ίσο με 0. Καλεί την ρουτίνα enter, η πρώτη εντολή είναι η tsl άρα «φέρνει» στο register το flag (0) και το flag γίνεται **1**. Μετά με την εντολή cmp συγκρίνω το register με το 0 (που είναι 0). Περνάει από την jnz χωρίς να κάνει τίποτα, αφού η συνθήκη δεν είναι ΑΛΗΘΗΣ, και φτάνει στο return όπου βγαίνει από την enter και μπαίνει στο Κ.Τ. της. Βγαίνοντας από το Κ.Τ. καλούμε την leave, **κάνουμε το flag 0**, ώστε να μπορέσει να μπει κάποια άλλη διεργασία στο Κ.Τ. της.



Αρχικά, ο καταχωρητής μπορεί και στη μια διεργασία και στην άλλη να έχει το ίδιο όνομα (π.χ. D5). Μπορεί, έτσι, να αναφερόμαστε στον ίδιο καταχωρητή αλλά σε διαφορετικό context (άλλα τα περιεχόμενα στη μια διεργασία, άλλα στην άλλη). *Θεωρούμε ότι ο καταχωρητής είναι ίδιος, αλλά με διαφορετικό context.*

Το flag είναι αρχικοποιημένο ως 0. Έρχεται να μπει η 1η διεργασία. Η πρώτη εντολή που θα εκτελέσει είναι η TSL. *Αν τύχει και «κοπεί» πριν την εκτελέσει είναι σαν να μη μπήκε ποτέ*. Θα μεταφέρει το περιεχόμενο του flag (0) στον καταχωρητή και θα κάνει το flag **1**. Έστω, σε αυτό το σημείο τυχαίνει και τελειώνει το κβάντο της άρα «κόβεται» η διεργασία*. Ό,τι έχει κάνει η διεργασία αποθηκεύεται στη μνήμη. Έρχεται η 2η διεργασία, «τρέχει» η εντολή tsl. Θα **μεταφέρει το περιεχόμενο του flag (1) στον καταχωρητή** και θα κάνει το flag 1. Πάει στην επόμενη εντολή, συγκρίνει τον καταχωρητή με το 0 (αλλά δεν είναι 0). Μετά, πάει στην εντολή jnz και αφού ο **καταχωρητής δεν έχει την τιμή 0 θα ξαναπάει στην εντολή tsl**. Μεταφέρει ξανά την τιμή του flag στον καταχωρητή και ξάνει το flag πάλι **1.** Συνεχίζει συγκρίνοντας τον καταχωρητή με το 0, πάλι στο jnz και ξανά από την αρχή. Θα κάνει συνέχεια αυτό το loop ώσπου να τελειώσει το κβάντο της. Θα «τρέξουν» τυχόν άλλες διεργασίες μέχρι που θα ξαναέρθει η σειρά της 1ης. Ξαναγυρνώντας σε αυτή όλα τα περιεχόμενα την CPU, ήρθαν από

την μνήμη στην CPU και η διεργασία συνεχίζει από εκεί που σταμάτησε σαν να μη πειράχτηκε τίποτα. Αφού είχε ήδη εκτελέσει την εντολή TSL προτού «κοπεί», συγκρίνει την τιμή του καταχωρητή (0) με τη τιμή 0. Πάει στην jnz αλλά δεν κάνει τίποτα καθώς είναι 0 και πάει παρακάτω στην ret και γυρνάει και μπαίνει στο Κ.Τ. της. Αν τώρα «κοπεί» μέσα στο Κ.Τ. της, πάλι η άλλη διεργασία δε θα μπορεί να μπει στο δικό της, αφού το flag θα είναι 1, συνεπώς θα παραμένει στην ίδια loop.
Το πρόβλημα με τις κοινές μεταβλητές που είχαμε προηγουμένως, λύνεται με την εντολή TSL. Όπως βλέπουμε, ο πρώτος που θα διαβάσει την flag θα την αλλάξει κιόλας πριν προλάβει να τη διαβάσει κάποιος άλλος. Αν, λοιπόν, κάποιος την βρει 0 είναι απόλυτα σίγουρο ότι ο επόμενος που θα την διαβάσει θα την βρει 1.

Πρόβλημα λύσης Peterson και TSL εντολής

Η λύση του Peterson και η TSL εντολή κάλυπταν αυτά που απαιτούσε μια καλή λύση. Γιατί όμως δεν είναι τόσο καλές; Η ενεργός αναμονή. Όσο μια διεργασία προσπαθεί να μπει στο Κ.Τ. της «σπαταλάει» κβάντα, καταναλώνει χρόνο της CPU (θα χρειαστεί να καταναλώνει κβάντα στο loop, μέχρι να μπει στο Κ.Τ. της)

Το πρόβλημα του Παραγωγού – Καταναλωτή

Έχουμε κάποιον (ή κάποιους) οι οποίοι παράγουν πράγματα και κάποιον (ή κάποιους) που τα καταναλώνουν. Ουσιαστικά αναφερόμαστε στο προγράμματα, που άλλα παράγουν δεδομένα και άλλα καταναλώνουν. Έχουμε δύο διαφορετικά προγράμματα που πρέπει να επικοινωνούνε, το ένα το ονομάζουμε Παραγωγό και το άλλο Καταναλωτή. Καταλαβαίνουμε ότι πρέπει να υπάρχει ένας συγχρονισμός ώστε ο παραγωγός να μη παράγει πάρα πολλά πράγματα πριν προλάβει να τα καταναλώσει ο καταναλωτής, γιατί δε θα έχουμε που να τα βάλουμε. Ο καταναλωτής, επίσης, να μην καταναλώνει πάρα πολύ γρήγορα τα πράγματα γιατί θα φτάσει σε ένα σημείο να μην έχει τίποτα να καταναλώσει. Επειδή όλοι οι πόροι στους υπολογιστές είναι πεπερασμένοι, δεν γίνεται να αφήσουμε τον παραγωγό να παράγει επ' άπειρον πράγματα. Έχουμε, λοιπόν, ένα χώρο στη μνήμη που ονομάζουμε buffer (απομονωτή) όπου εκεί θα τοποθετεί τα δεδομένα του ο παραγωγός και παράλληλα από εκεί θα τα παίρνει και ο καταναλωτής. Πρέπει να φροντίσουμε ώστε όταν ο buffer γεμίζει, ο παραγωγός να μην παράγει άλλα δεδομένα και αν αδειάσει ο καταναλωτής να μη πάει να πάρει κάτι από κει μέσα.

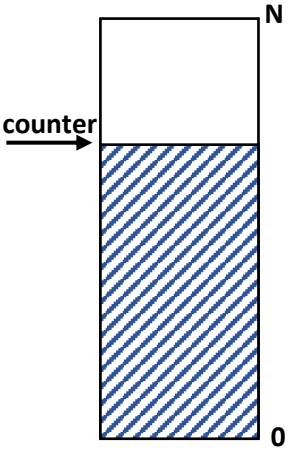
Παραγωγός

```
#define N 100
int count=0
producer() {
    int item
    while (TRUE) {
        produce_item(item)
        if (count=N) sleep()
        enter_item(item)
        count=count+1
        if (count=1) wakeup(consumer)
        }
    }
```

Καταναλωτής

```
consumer() {
    int item
    while (TRUE) {
        if (count=0) sleep()
        remove_item(item)
        count=count-1
        if (count=N-1) wakeup(producer)
        consume_item(item)
        }
    }
```

Buffer



Αρχικά, θεωρούμε ότι ο buffer έχει μέγεθος N, δηλαδή 100. Έχουμε και έναν ακέραιο μετρητή (counter), που ξεκινάει από το 0 και θα λέει πόσο γεμάτος είναι ο buffer. Το N και το count αποτελούν global μεταβλητές, αφορούν δηλαδή όλα τα προγράμματα. ***Εδώ να σημειωθεί, ότι οι ρουτίνες producer() και consumer() αποτελούν δύο διαφορετικές διεργασίες.*** Ξεκινώντας με τη ρουτίνα του παραγωγού, ορίζουμε το item ως ακέραιο (θα μπορούσε να είναι και άλλου τύπου π.χ. float, αλλά δεν έχει σημασία) και συμβολίζει ουσιαστικά το κομμάτια δεδομένων που παράγει. Έπειτα ακολουθεί ένα while loop, που εξ ορισμού η συνθήκη είναι ΑΛΗΘΗΣ, οπότε συνεχώς παράγονται δεδομένα. Η πρώτη εντολή μέσα στο loop είναι μια ρουτίνα produce_item(item) που παράγει κάποιο δεδομένο. Αφού το παράξει, θα προσπαθήσει να το τοποθετήσει στον buffer. Το πρώτο που ελέγχει με την εντολή if (count=N) είναι να δει αν ο buffer έχει γεμίσει. Σε περίπτωση που η συνθήκη είναι ΑΛΗΘΗΣ, δηλαδή ο buffer είναι γεμάτος, καλεί την ρουτίνα sleep() η οποία αυτό το οποίο κάνει είναι να πάρει τη διεργασία και να την βάλει σε κατάσταση αναμονής. Αν τώρα περάσει από τη συνθήκη, καλεί την ρουτίνα enter_item που βάζει τα δεδομένα στο buffer και μετά αυξάνει το count κατά ένα. Τέλος, ελέγχει αν το count ισούται με ένα και καλεί την ρουτίνα wakeup() για να «ξυπνήσει» τον καταναλωτή. Αυτό γίνεται διότι στη περίπτωση που ο buffer είναι άδειος ο καταναλωτής δε θα είχε τίποτα «να πάρει» από μέσα, άρα το πιο πιθανό είναι «να κοιμάται». Επομένως, το ΛΣ θα πάρει τον καταναλωτή από την κατάσταση WAIT και θα τον φέρει στην κατάσταση READY και μπορεί πλέον να «τρέξει» όταν έρθει η ώρα του.

Βλέπουμε ότι η ρουτίνα consumer() είναι αντίστοιχη με αυτή του παραγωγού, απλά έχει κάποιες μικροαλλαγές. Ορίζουμε το item ως ακέραιο (θα μπορούσε να είναι και άλλου τύπου π.χ. float, αλλά δεν έχει σημασία) και συμβολίζει ουσιαστικά το κομμάτια δεδομένων που καταναλώνει. Έπειτα ακολουθεί ένα while loop, που εξ ορισμού η συνθήκη είναι ΑΛΗΘΗΣ, οπότε συνεχώς καταναλώνονται δεδομένα. Το πρώτο που ελέγχουμε είναι να μην είναι άδειος ο buffer (count = 0). Σε περίπτωση που είναι άδειος, καλούμε την ρουτίνα sleep() όπως και στην πάνω περίπτωση. Εάν τώρα το count δεν είναι 0 και περάσει στην επόμενη εντολή, καλεί τη ρουτίνα remove_item(item), η οποία αφαιρεί από το buffer το item (τα δεδομένα). Στη συνέχεια, μειώνει το count κατά ένα. Έπειτα, ελέγχει αν ο buffer δεν είναι πλήρης (count=N-1) και καλεί τη ρουτίνα wakeup() για να «ξυπνήσει» τον παραγωγό. Αυτό γίνεται διότι στη περίπτωση που ο buffer είναι γεμάτος ο παραγωγός δε θα είχε χώρο να παράξει νέα δεδομένα, άρα το πιο πιθανό είναι «να κοιμάται». Επομένως, το ΛΣ θα πάρει τον παραγωγό από την κατάσταση WAIT και θα τον φέρει στην κατάσταση READY και μπορεί πλέον να «τρέξει» όταν έρθει η ώρα του. Στη συνέχεια καλείται η υπορουτίνα consume_item(item) που χρησιμοποιεί/επεξεργάζεται τα δεδομένα.

<u>Πρόβλημα – κοινή μεταβλητή count</u>

```
#define N 100
int count=0
producer() {
    int item
    while (TRUE) {
        produce_item(item)
        if (count=N) sleep()
        enter_item(item)
        count=count+1
        *if (count=1) wakeup(consumer)
        }
    }
```

```
consumer() {
    int item
    while (TRUE) {
        if (count=0) sleep() *2
        remove_item(item)
        count=count-1
        if (count=N-1) wakeup(producer)
        consume_item(item)
        }
    }
```

Έστω ότι το count δεν είναι 0, οπότε μετά την αύξηση δεν ισούται με 1, άρα δε θα προσπαθήσει να ξυπνήσει τον καταναλωτή. Υποθέτουμε, λοιπόν, ότι μετά την αύξηση τελειώνει το κβάντο και

«κόβεται» *. Έρχεται και τρέχει ο καταναλωτής, συνεχίζει μέχρι το count να φτάσει στο 0, οπότε καλείται η ρουτίνα sleep() και «κοιμάται»*2. Όταν έρθει η σειρά του παραγωγού και συνεχίσει από εκεί που διακόπηκε, δε θα ξυπνήσει τον καταναλωτή, αφού το count δεν είναι 1. Συνεχίζει να «τρέχει» και βάζει μέσα στο buffer συνεχώς πράγματα χωρίς να ξυπνήσει τον καταναλωτή αφού το count δε το βλέπει ίσο με 1. Θα συνεχίσει λοιπόν μέχρι το count να γίνει Ν, δηλαδή να γεμίσει ο buffer και θα καλέσει τη ρουτίνα sleep(), οπότε και ο παραγωγός θα πάει «να κοιμηθεί». *Τι γίνεται λοιπόν;* Δε θα «ξυπνήσει» ποτέ ο καταναλωτής, «πάει για ύπνο» και ο παραγωγός. Όπως ξαναείδαμε όταν έχουμε κοινή μεταβλητή και δύο διεργασίες πάνε να την προσπελάσουν θα έχουμε πρόβλημα.

Λύση; Σηματοφορείς (semaphores)

Σηματοφορείς (semaphores)

Εφευρέθηκαν τη δεκατία του ΄60 από τον Djikstra και την ομάδα του. Είναι ειδικές μεταβλητές που τις χειρίζεται το ΛΣ. Παρακάτω βλέπουμε τις δύο συναρτήσεις down και up που χειρίζονται τα semaphores.

```
down(s)
   { if s>0 then
     s=s-1
   else
     sleep
   endif
   }

up(s)
   { s=s+1
   wake_a_process
   }
```

Ορίζουμε έστω ως s ένα semaphore.
Τι κάνει η down; Ελέγχει αρχικά να δει αν το semaphore είναι μεγαλύτερο το 0. Αν η συνθήκη είναι ΑΛΗΘΗΣ το μειώνει κατά ένα. Αν δεν είναι μεγαλύτερη του 0 (άρα είναι 0), πάει στο sleep. *Τι κάνει η up;* Αυξάνει το semaphore κατά 1 και ξυπνάει μια διεργασία. *Τι διαφορά έχει η down και η up με τα προηγούμενα που είδαμε;* Και οι δύο αποτελούν συναρτήσεις του ΛΣ, οπότε δεν μπορούν να διακοπούν στη μέση. Άρα θα εκτελεστούν ΟΛΟΚΛΗΡΕΣ και δεν μπορούν να κοπούν ενδιάμεσα.

Το ΛΣ κρατάει μια λίστα με τα semaphores και για κάθε ένα κρατάει και μια λίστα με διεργασίες που έχουν κοιμηθεί λόγω του αντίστοιχου semaphore. Άρα όσες διεργασίες προσπάθησαν να τρέξουν το down και φτάσανε στο sleep γιατί το s το βρήκανε 0, είναι όλες στη λίστα του αντίστοιχου semaphore. Όταν λοιπόν κάποια άλλη διεργασία προσπαθήσει να ξυπνήσει μία από αυτές τις διεργασίες, τότε το ΛΣ θα τις ξυπνήσει είτε όλες, είτε την πρώτη που κοιμήθηκε είτε ακολουθώντας όποιον άλλο αλγόριθμο θέλει. Το να τις ξυπνήσει όλες, δεν είναι κακό. Μια διεργασία που «κοιμήθηκε» όταν ξυπνήσει θα επαναλάβει την συνάρτηση down, θα τρέξουν δηλαδή από την αρχή ΟΛΕΣ οι εντολές.

Ο μόνος τρόπος για να περάσει μια διεργασία από την down, είναι το semaphore να είναι μεγαλύτερο του 0, αλλιώς κάθε φορά θα πηγαίνει στο sleep.

Ο μόνος τρόπος που μπορούμε να χειριστούμε τα semaphores είναι με το up και το down! Δεν μπορούμε εμείς οι ίδιοι να βλέπουμε ή αναθέτουμε τιμές σε αυτά.

Παραγωγός – Καταναλωτής με σηματοφορείς

```
semaphore mutex=1, empty=N, full=0
producer() {
    int item
    while (TRUE) {
        produce_item(item)
        down(empty)
        down(mutex)
        enter_item (item)
        up(mutex)
        up(full)
        }
    }
```

Προστασία Κ.Τ.

```
consumer()
    { int item
    while (TRUE)
        {
        down(full)
        down(mutex)
        remove_item(item)
        up(mutex)
        up(empty)
        consume_item(item)
        }
    }
```

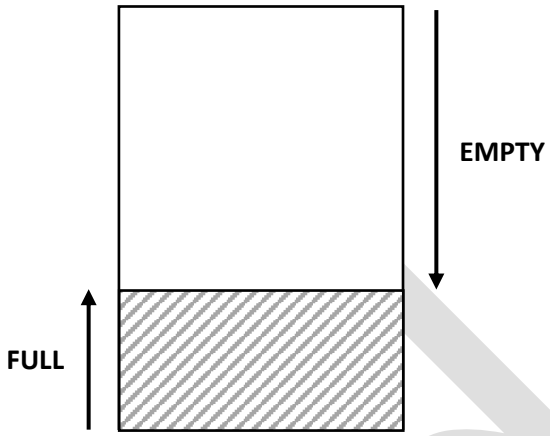
Προστασία Κ.Τ.

Βλέπουμε πως οι σηματοφορείς δηλώνονται με τον ειδικό τύπο μεταβλητής **semaphore**. Το μόνο που μπορούμε να κάνουμε, είναι να του δώσουμε αρχική τιμή. Πέρα από αυτό μέσα στο πρόγραμμα δε μπορούμε ούτε να το ελέγξουμε ούτε να το αλλάξουμε, παρά μόνο με τη χρήση του up και του down. Αρχικοποιούμε 3 semaphores, το mutex (**mut**ual **ex**clusion ~ αμοιβαίος αποκλεισμός ~) που αποτελεί ένα binary semaphore (παίρνει δηλαδή ως τιμές μόνο το 0 και το 1) με 1, το empty (ίσο με N) και το full (ίσο με 0) που βοηθούν στο να γνωρίζουμε πότε είναι γεμάτος ή άδειος ο buffer. Η αρχική τιμή του empty μας δείχνει και το μέγεθος του buffer. Ξεκινάει λοιπόν η ρουτίνα του παραγωγού, ορίζουμε έναν item ακέραιο τύπου όπως προηγουμένως και πάμε στο while loop με την εξ ορισμού συνθήκη TRUE. Αρχικά με την produce_item() παράγεται αυτό που θέλω να βάλω στο buffer. Μετά βλέπουμε το down(empty). Αν το empty είναι 0, «κοιμάται», δηλαδή ο παραγωγός θα κολλήσει σε εκείνη την εντολή. Από το πρώτο down, θα περάσει μόνο όταν μπορέσει να μειώσει το empty κατά ένα. Δηλαδή στη περίπτωση που το empty είναι ίσο με το N, θα «τρέξει» η διεργασία παράγοντας δεδομένα ώσπου το empty να γίνει 0 και να «κολλήσει» στην εντολή down(empty). Έστω ότι δεν είναι 0, το μειώνουμε κατά ένα και προχωράμε παρακάτω. Σε αυτό το σημείο ας τονιστεί ότι θεωρούμε το γράψιμο/σβήσιμο από το buffer ως κρίσιμο τμήμα. Με λίγα λόγια, όταν π.χ. έρθει ο παραγωγός και γράψει τα 10 γράμματα από μια λέξη και «κοπεί» αν έρθει μετά ο καταναλωτής θα διαβάσει τα 10 πρώτα, άρα μόνα αυτά θα είναι σωστά, τα υπόλοιπα δε θα είναι. Για να μη προκύπτουν τέτοια προβλήματα λέμε ότι όταν κάποιος γράφει στον buffer, κάποιος άλλος δε διαβάζει και όταν κάποιος διαβάζει από buffer, κάποιος άλλος δε γράφει. Αυτό μας βοηθάει να το αντιμετωπίσουμε το semaphore mutex. Αρχική τιμή έχει 1, άρα όταν περάσει μια διεργασία από το down(mutex) και πάρει την τιμή 0, δε μπορεί να περάσει και $2^{η}$ διεργασία αφού θα βρίσκει το mutex 0 οπότε θα καταλήγει σε sleep. Αφού μπει και γράψει ότι χρειάζεται στον buffer, έχουμε το up(mutex) που θα το αυξήσει από 0 σε 1 και άρα πλέον θα μπορεί να μπει και άλλη διεργασία. Εφόσον έχουμε τοποθετήσει κάτι στον buffer, καλείται το up(full), αυξάνω δηλαδή το semaphore full κατά 1.

Περνώντας στον καταναλωτή, ορίζουμε έναν item ακέραιο τύπου όπως προηγουμένως και πάμε στο while loop με την εξ ορισμού συνθήκη TRUE. Το πρώτο που θα κάνει είναι να προσπαθήσει να κάνει down το full, να μειώσει δηλαδή το semaphore full κατά 1. Σε περίπτωση που είναι 0, δηλαδή ο buffer είναι άδειος, θα πέσει σε sleep και θα «κολλήσει» εκεί πέρα. Αρχικοποιήσαμε το full ως 0, οπότε σε περίπτωση που «τρέξει» πρώτα ο καταναλωτής αντί για τον παραγωγό, θα πέσει πάνω στο sleep. Περνώντας από εκεί, σημαίνει ότι δεν είναι άδειος ο buffer, θα προσέξει να είναι η μοναδική εργασία που εκείνη τη στιγμή πάει να διαβάσει κάτι από το buffer και αυτό θα το πετύχει με τη χρήση του down(mutex). Αν το semaphore mutex είναι ίσο με 0, θα «κολλήσει» εκεί και δε θα προχωρήσει. Αν είναι 1, θα το κάνουμε 0 και περνάμε παρακάτω. Διαβάζουμε αυτό που είναι να διαβάσουμε από τον buffer και έπειτα καλούμε το up(mutex) ώστε από 0 να γίνει 1 και να μπορεί

κάποια άλλη διεργασία να μπει στο Κ.Τ. της και να γράψει/διαβάσει στον buffer. Αμέσως μετά, καλείται το up(empty) αφού εφόσον έχουμε πάρει κάτι από το buffer πρέπει να αυξήσουμε τον άδειο χώρο. Άρα το semaphore empty αυξάνεται κατά 1. Τέλος, καταναλώνεται αυτό που διαβάσαμε.

Σχηματική απεικόνιση buffer:



Αναγνώστες – συγγραφείς

Υποθέτουμε ότι έχουμε μια βάση δεδομένων που κάποιοι πάνε και γράφουν και άλλοι πάνε και διαβάζουν στοιχεία από αυτήν. Έχουμε δηλαδή συγγραφείς και αναγνώστες. Το πρόβλημα μπορεί να θυμίζει το παραπάνω με τον παραγωγό – καταναλωτή, αλλά σε αυτήν την περίπτωση δεν έχουμε κάποιο buffer, καθώς μιλάμε για μια βάση δεδομένων της οποίας το μέγεθος μπορεί κάλλιστα να μεγαλώσει. Αυτό που θέλουμε να πετύχουμε είναι όταν γράφει κάποιος στην βάση, να μη μπορεί κανείς άλλος ούτε να γράψει ούτε να διαβάσει. Από την άλλη, όταν διαβάζει κάποιος, μπορεί να μπει κι άλλος να διαβάσει. Άρα μέσα στη βάση είτε θα υπάρχει ΑΠΟΚΛΕΙΣΤΙΚΑ ΕΝΑΣ συγγραφέας είτε ένας ή πολλοί αναγνώστες (προφανέστατα όταν είναι μέσα οι αναγνώστες, δεν μπορεί να μπει κάποιος συγγραφέας).

Αναγνώστης

```
semaphore mutex=1, db=1
int rc=0
reader()
    { while (TRUE) {
        down(mutex) *3          ┐ Προστασία Κ.Τ.
        rc=rc+1                 │
        if (rc=1) down(db) *2   │
        up(mutex)               ┘
        read_database()
        down(mutex)             ┐ Προστασία Κ.Τ.
        rc=rc-1                 │
        if (rc=0) up(db)        │
        up(mutex)               ┘
        use_data()
    }}
```

Συγγραφέας

```
writer()
   {
   while (TRUE)
      {
      think_up_data()
      down(db) *
      write_database()
      up(db)
      }
   }
```

Αρχικά, χρειαζόμαστε λιγότερα semaphores από πριν, το mutex (αρχική τιμή 1) και ένα db (αρχική τιμή 1) για την ΒΔ. Και τα δύο θα είναι binary semaphores (0 ή 1). Το mutex θα δείχνει αν μπορώ να μπω στο Κ.Τ. ή όχι, ενώ το db αντίστοιχα για την ΒΔ. Ακόμα έχουμε μια ακέραιου τύπου κοινή μεταβλητή rc (reader count) που θα δείχνει πόσοι είναι αναγνώστες κάθε φορά (αρχική τιμή 0). *Γιατί να μην ήταν και το rc semaphore;* Στα semaphores μόλις φτάνω στο 0, δε μπορώ να προχωρήσω παρακάτω και «πέφτει για ύπνο» η διεργασία. Ξεκινώντας με τη ρουτίνα του αναγνώστη, υπάρχει πάλι ένα while loop (εξ ορισμού TRUE). Πρώτον, θα γίνει down το mutex, γιατί θα προσπαθήσω να αλλάξω τον αριθμό των αναγνωστών (Κ.Τ.). Αυξάνω το rc κατά 1, μετά βλέπω ότι σε περίπτωση που το rc είναι 1 (δηλαδή είμαι ο πρώτος αναγνώστης) πρέπει να ελέγξω ότι δεν υπάρχει κάποιος συγγραφέας μέσα. *Αν δεν είμαι ο πρώτος, εφόσον υπάρχουν άλλοι αναγνώστες έχει εξασφαλιστεί ότι δεν υπάρχει συγγραφέας μέσα και μπαίνω και γω.* Έτσι αν το rc=1, βάζω down(db) που σημαίνει ότι αν υπάρχει συγγραφέας μέσα, δηλαδή το db=0, δε θα μπορέσω να περάσω από εκεί γιατί θα πέσω στο sleep. Αν περάσω από το down(db), το κάνω δηλαδή 0, προχωράω κάνοντας up το mutex (αφού βγαίνω από Κ.Τ.) και διαβάζω από τη ΒΔ. Αν έρθει και 2ος αναγνώστης θα μπει «στα τυφλά» να διαβάσει από την ΒΔ, καθώς θα αυξήσει το rc κατά ένα (θα γίνει δηλαδή 2) και δε θα κάνει down(db). Συνεχίζοντας για να βγω, πρέπει να μειώσω το rc κατά ένα, άρα θα πρέπει να το προστατεύσω πάλι με semaphores. Στη περίπτωση που είναι το rc=0, σημαίνει ότι είμαι ο τελευταίος αναγνώστης, άρα πρέπει να κάνω up(db) ώστε να μπορέσει να μπει κάποιος συγγραφέας αν θελήσει.
Η ρουτίνα του συγγραφέα είναι σχετικά πιο απλή. Έχουμε, πάλι ένα while loop (εξ ορισμού TRUE). Ξεκινάμε με την υπορουτίνα think_up_data(), που αποτελεί τη ρουτίνα που «σκέφτεται» τα δεδομένα (π.χ. πράξεις δεδομένων που θα βάλουμε μέσα στη ΒΔ). Έπειτα, έχουμε πάλι semaphores. Υπάρχει το down(db) που χρησιμοποιούμε πριν μπούμε να γράψουμε στη ΒΔ. Καταλαβαίνουμε, ότι στη περίπτωση που υπάρχει κάποιος αναγνώστης στη ΒΔ, το db θα είναι 0, οπότε με το down(db) συγγραφέας θα πέσει στο sleep και θα «κολλήσει» εκεί. Αν τώρα έχει μπει άλλος συγγραφέας, πάλι το db θα είναι 0, οπότε και πάλι θα «κολλήσουμε» εκεί. Στη περίπτωση που είναι 1, το db από 1 θα το κάνει 0, θα μπει να γράψει στη ΒΔ και βγαίνοντας θα κάνει το db πάλι 1 με το up(db).

Όταν ένας συγγραφέας είναι ήδη στη ΒΔ και έρθει ένας άλλος θα «κολλήσει» στο down(db)*. Όταν έρθει ο πρώτος αναγνώστης θα «κολλήσει» και αυτός στο down(db)*2, ενώ ο κάθε επόμενος θα «κολλάει» στο down(mutex)*3.

Συγχρονισμός Διεργασιών

** Το loop … until (forever) παρόμοιο με το while (TRUE)*

```
process a()
loop
    {
    …
    …
    event
    …
    …
    }
until (forever)
```

```
process b()
loop
    {
    …
    …
    event
    …
    …
    }
until (forever)
```

Θέλουμε να συγχρονίσουμε δύο διεργασίες έτσι ώστε τα events να γίνονται με μια χρονική σειρά. Οι διεργασίες αυτές είναι ανεξάρτητες η μία από την άλλη αλλά και ασύγχρονες (όποτε έρθει στη καθεμία κβάντα «τρέχει»). Για να αποτρέψω να εκτελεστεί κάποιο γεγονός εκτός σειράς, χρησιμοποιώ πριν από αυτό κάποιο down.

Έστω ότι τα θέλω έτσι: **1 2 1 2 1 2 1 2 1 2**

Λύση στο παραπάνω:

```
Semaphore P1 = 1, P2 = 0;
process a()
loop
    {
    …
    …
    down(P1)
    event 1
    up(P2)
    …
    …
    }
until (forever)
```

```
process b()
loop
    {
    …
    …
    down(P2)
    event 2
    up(P1)
    …
    …
    }
until (forever)
```

Έστω ότι τα θέλω έτσι: **1 2 2 1 2 2 1 2 2**

Λύση στο παραπάνω:

```
Semaphore P1 = 2, P2 = 0;
process a()
loop
    {
    …
    down(P1)
    down(P1)
    event 1
    up(P2)
    up(P2)
    …
    …
    }
until (forever)
```

```
process b()
loop
    {
    …
    …
    down(P2)
    event 2
    up(P1)
    …
    …
    }
until (forever)
```

Έστω ότι τα θέλω έτσι: **1 2 3 1 2 3 1 2 3** *(παράδειγμα από Open Courses)*

Semaphore S1 = 1, S2 = 0, S3 = 0;

```
process a()
loop
    {
    …
    …
    down(S1)
    event 1
    up(S2)
    …
    …
    }
until (forever)
```

```
process b()
loop
    {
    …
    …
    down(S2)
    event 2
    up(S3)
    …
    …
    }
until (forever)
```

```
process c()
loop
    {
    …
    …
    down(S3)
    event 3
    up(S1)
    …
    …
    }
until (forever)
```


Χρήσιμα Links:

- https://youtu.be/K6u1A5_5zV8?si=1OHhtpoERFsp-s9d
- https://www.javatpoint.com/os-tsl-mechanism
- https://www.geeksforgeeks.org/semaphores-in-process-synchronization/
- https://www.javatpoint.com/producer-consumer-problem-in-os
- https://www.geeksforgeeks.org/producer-consumer-problem-using-semaphores-set-1/
- https://www.geeksforgeeks.org/readers-writers-problem-set-1-introduction-and-readers-preference-solution/