ΜΥΥ601 Λειτουργικά Συστήματα

Εαρινό 2024

Μάθημα 5

Ταυτοχρονισμός: Συγχρονισμός

Τμήμα Μηχανικών Η/Υ & Πληροφορικής
Πανεπιστήμιο Ιωαννίνων

1

Περίγραμμα

- Σημαφόροι
- Παρατηρητές
- Μεταβίβαση μηνυμάτων
- Πρόβλημα Αναγνωστών-Γραφέων

2

Σημαφόροι (Dijkstra 1965)

- Επιτρέπουν τις διεργασίες να συνεργαστούν
  - Υποστηρίζουν συνεργασία με χρήση απλών σημάτων
  - Μια διεργασία σταματάει σε συγκεκριμένη περιοχή ΚΑΙ περιμένει μέχρι να λάβει συγκεκριμένο σήμα
- Σημαφόρος
  - Ειδική μεταβλητή που αναπαριστά ένα σήμα π.χ. σημαφόρος s
- *signal(s)*
  - Καλείται για την αποστολή σήματος μέσω της σημαφόρου s
- *wait(s)*
  - Καλείται για την λήψη σήματος μέσω σημαφόρου s
  - Η διεργασία περιμένει μέχρι την αποστολή του σήματος

Αναπαράσταση

- Δείτε τη σημαφόρο σαν μια ακέραιη μεταβλητή
  - Μια σημαφόρος μπορεί να αρχικοποιηθεί σε τιμή ≥ 0
  - wait() μειώνει την τιμή της σημαφόρου κατά 1
    - Αν η τιμή γίνει <0, η διεργασία που κάλεσε wait() $\Rightarrow$ αποκλεισμό
  - signal() αυξάνει την τιμή της σημαφόρου κατά 1
    - Αν η τιμή γίνει ≤ 0, αίρεται ο αποκλεισμός σε μια διεργασία
  - signal()/wait(): ατομικές λειτουργίες
  - Δεν υπάρχει άλλος τρόπος να δούμε/αλλάξουμε τη σημαφόρο
- Δυαδική σημαφόρος
  - Ειδική περίπτωση σημαφόρου που παίρνει τιμές $\in \{0, 1\}$
  - Αποδεικνύεται ότι έχει την εκφραστική δύναμη των γενικών σημαφόρων

Ορισμός Σημαφόρου

```
struct semaphore {
  int count;
  queueType queue;
}
```

```
void wait(semaphore s) {
  s.count--;
  if (s.count < 0) {
    place process in s.queue;
    block this process;
  }
}
```

```
void signal(semaphore s) {
  s.count++;
  if (s.count <=0) {
    remove a process P from
          s.queue;
    place process P on ready list;
  }
}
```


5

Ορισμός Δυαδικής Σημαφόρου

```
struct binary_semaphore {
  enum value {0, 1};
  queueType queue;
}
```

```
void waitB(binary_semaphore s) {
  if (s.value == 1)
    s.value = 0;
  else {
    place this process in s.queue;
    block this process;
  }
}
```

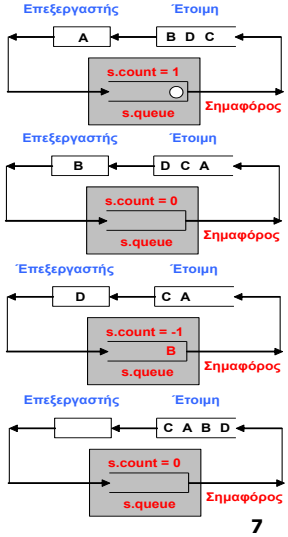
```
void signalB(binary_semaphore s) {
  if (s.queue.is_empty())
     s.value = 1;
  else {
    remove a process P from
          s.queue;
    place process P on ready list;
  }
}
```


6

Παράδειγμα

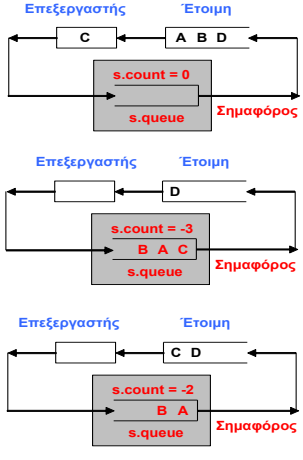
- Τέσσερις διεργασίες *A, B, C, D*
  - *A, B, C* καταναλώνουν δεδομένα
  - *D* παράγει δεδομένα
- Η σημαφόρος *s* μετράει δεδομένα

1. *A* καταναλώνει διαθέσιμο στοιχείο
2. *B* καλεί wait(s) και μπαίνει στην s.queue
3. *D* καλεί signal(s) και απομακρύνει την *B* από την s.queue



Παράδειγμα (συνέχεια)

1. *C* καλεί wait(s) και μπαίνει στην s.queue
2. *A* καλεί wait(s) και μπαίνει στην s.queue
3. *B* καλεί wait(s) και μπαίνει στην s.queue
4. *D* καλεί signal(s) και απομακρύνει την *C* από την s.queue

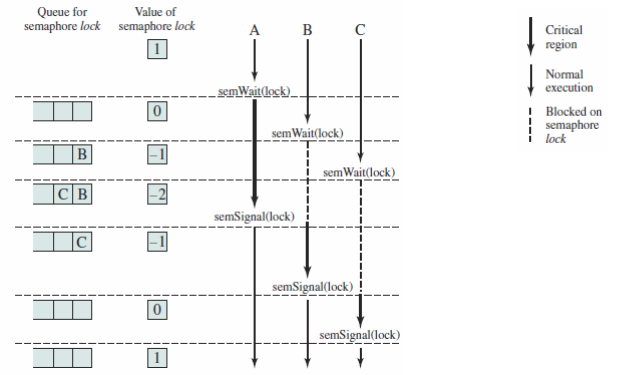


Αμοιβαίος Αποκλεισμός με Σημαφόρους

- Θεωρούμε n διεργασίες
  - P(1), ..., P(n)
- Αρχικοποίηση σημαφόρου
  - *mutual_exclusion* = 1
- wait(s)
  - Η πρώτη διεργασία μπαίνει στην κρίσιμη περιοχή
  - Οι άλλες μειώνουν τη *mutual_exclusion* κατά 1 και μπαίνουν σε αποκλεισμό
- signal(s)
  - Καλείται όταν μια διεργασία βγαίνει από την κρίσιμη περιοχή
  - Αυξάνει τη *mutual_exclusion* κατά 1
  - Αίρει τον αποκλεισμό σε μια διεργασία
  - Την επιτρέπει να εισέλθει στην κρίσιμη περιοχή όταν δρομολογηθεί

```
/* mutual exclusion program */
const int n = /* # processes */;
semaphore mutual_exclusion = 1;
void P(int i) {
    while (true) {
        wait(mutual_exclusion);
        /* critical section */
        signal(mutual_exclusion);
        /* remainder */;
    }
}
void main()
    {parbegin(P(1),...,P(n));}
```


Παράδειγμα Αμοιβαίου Αποκλεισμού

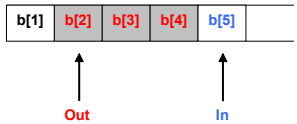


Παρατηρήσεις στον Αμοιβαίο Αποκλεισμό

- Αρχική τιμή της s.count
  - Το πλήθος των διεργασιών που επιτρέπεται να εισέλθουν στην κρίσιμη περιοχή
- s.count ≥ 0
  - |s.count|: πλήθος διεργασιών που μπορούν να καλέσουν wait() χωρίς να χρειαστεί να μπουν στην s.queue
- s.count < 0
  - |s.count|: πλήθος διεργασιών που περιμένουν στην s.queue

Πρόβλημα Παραγωγού-Καταναλωτή

- Κοινόχρηστη ενδιάμεση μνήμη
  - Περιέχει αντικείμενα κάποιου τύπου
  - Ένας ή περισσότεροι παραγωγοί εισάγουν
  - Ένας καταναλωτής απομακρύνει
  - Πράκτορας είναι κάθε παραγωγός ή καταναλωτής
- Περιορισμοί
  - Ένας πράκτορας έχει πρόσβαση κάθε φορά
  - Πρέπει να ισχύει Out <= In



Απεριόριστη ενδιάμεση μνήμη
για το πρόβλημα παραγωγού-καταναλωτή

```
producer:
while (true) {
    /* produce item v */
    b[in] = v;
    in++;
}
consumer:
while (true) {
    while (in <= out)
/* empty - do nothing */;
    w = b[out];
    out++;
    /* consume item w */
}
```

Λύση με Δυαδικές Σημαφόρους

- Ακέραιος *n*
  - Το πλήθος των αντικειμένων στη μνήμη (= In – Out)
  - Αυξάνεται από τον παραγωγό
  - Μειώνεται από τον καταναλωτή
- Δυαδική σημαφόρος *mutual_exclusion*
  - Επιτρέπει έναν πράκτορα κάθε φορά να έχει πρόσβαση
- Δυαδική σημαφόρος *full_slots*
  - Ο καταναλωτής περιμένει όταν βρίσκει τη μνήμη άδεια
  - Ο παραγωγός σηματοδοτεί όταν εισάγει σε άδεια μνήμη

Implementation

```
/* Λύση παραγωγού καταναλωτή */
int n = 0;
binary_semaphore mutual_exclusion =1;
binary_semaphore full_slots = 0;
void producer() {
    while (true) {
        produce();
        waitB(mutual_exclusion);
        append();
        n++;
        if (n == 1)
            signalB(full_slots);
        signalB(mutual_exclusion);
    }
}
```

```
void consumer() {
    waitB(full_slots);
    while (true) {
        waitB(mutual_exclusion);
        take();
        n--;
        signalB(mutual_exclusion);
        consume()
        if (n==0)
            waitB(full_slots);
    }
}
void main() {
  parbegin(producer, consumer);
}
```

Buffer Underflow Scenario

Παραγωγός

3. Βρίσκει n = 0
4. Θέτει n = n + 1 = 1 [n++]
5. Καλεί signal() όταν full_slots = 1
   [if (n = = 1) signalB(full_slots)]

Καταναλωτής

1. Βρίσκει n = 1
2. Θέτει n = n – 1 = 0 [n--]
6. Βρίσκει n = 1 (καθόλου αναμονή)
   [if (n = = 0) waitB(full_slots)]
7. Θέτει n = n – 1= 0 [n--]
8. Καλεί wait() όταν full_slots = 1
   [if (n = = 0) waitB(full_slots)]
9. Θέτει full_slots = 0(χωρίς αναμονή)
10. Θέτει n = n – 1 = -1 [n--]
    Σφάλμα κάτω υπερχείλισης !

Πρόβλημα και Πιθανή Προσέγγιση

- Κανονικά
  - Ο καταναλωτής βρίσκει *n = 0* και περιμένει για σήμα *full_slots*
  - Ο παραγωγός εισάγει αντικείμενο στην άδεια ενδιάμεση μνήμη, θέτει *n = 1* και σηματοδοτεί τον καταναλωτή μέσω της *full_slots*
- Πρόβλημα
  - Ο παραγωγός θέτει *n = 1* και καλεί *signal(full_slots)*
  - Ο καταναλωτής βρίσκει *n = 1* και ΔΕΝ καλεί *wait(full_slots)*
  - *…*
  - Το αχρησιμοποίητο *full_slots = 1* οδηγεί σε κάτω υπερχείλιση
- Πιθανή προσέγγιση
  - Μετακινούμε [if (n==0) waitB(full_slots)] εντός κρίσιμης περιοχής
  - Μπορεί να οδηγήσει σε αδιέξοδο όταν ο καταναλωτής καλεί waitB() !

Λύση

- Εισάγουμε την τοπική μεταβλητή *m* στον καταναλωτή
- Προσθέτουμε *m = n* μετά την εντολή *n--*
- Έτσι αναγκάζουμε τον καταναλωτή να δει τη μετάβαση του *n* μέσω του *0* με την εντολή [if (m==0)]

Σωστή Υλοποίηση

```
/* producer consumer solution */
int n = 0; /* count items */
binary_semaphore mutual_exclusion=1;
binary_semaphore full_slots = 0;
void producer() {
    while (true) {
        produce();
        waitB(mutual_exclusion);
        append();
        n++;
        if (n == 1)
            signalB(full_slots);
        signalB(mutual_exclusion);
    }
}
```

```
void consumer() {
    int m; /* local variable */
    waitB(full_slots);/*first item*/
    while (true) {
        waitB(mutual_exclusion);
        take();
        n--; m = n;
        signalB(mutual_exclusion);
        consume()
        if (m==0)
            waitB(full_slots);
    }
}
void main() { parbegin(producer,
    consumer); }
```

Λύση με Γενικές Σημαφόρους

- Δηλώνουμε το full_slots ως γενική σημαφόρο (ή *απαρίθμησης*)
  - Ο παραγωγός καλεί *signal(full_slots)* όταν προσθέτει αντικείμενο
  - Ο καταναλωτής καλεί *wait(full_slots)* όταν αφαιρεί αντικείμενο
  - *full_slots.count* ≤ 0 υποδηλώνει άδεια μνήμη
- Παραγωγός
  - *signal(full_slots)* ακολουθεί την κρίσιμη περιοχή [wait(s)…signal(s)]
- Καταναλωτής
  - *wait(full_slots)* προηγείται της κρίσιμης περιοχής [wait(s)…signal(s)]

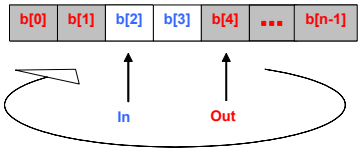
Υλοποίηση με Γενικές Σημαφόρους

```
/* producer consumer solution */
semaphore full_slots=0;
semaphore mutual_exclusion=1;

void producer() {
    while (true) {
        produce();
        wait(mutual_exclusion);
        append();
        signal(mutual_exclusion);
        signal(full_slots);
    }
}
```

```
void consumer() {
    while (true) {
        wait(full_slots);
        wait(mutual_exclusion);
        take();
        signal(mutual_exclusion);
        consume();
    }
}

void main() {
  parbegin(producer, consumer);
}
```

Παραγωγός-Καταναλωτής Οριοθετημένης Μνήμης

- Οριοθετημένη μνήμη
  - Κυκλικός πίνακας μεγέθους n
  - Ο δείκτης γίνεται 0 μετά το n-1 (mod n)
- Παραγωγός
  - Σε αποκλεισμό όταν γεμίζει η μνήμη
  - Άρση με απομάκρυνση από καταναλωτή
- Καταναλωτής
  - Σε αποκλεισμό όταν αδειάζει η μνήμη
  - Άρση όταν εισάγει ο παραγωγός



Οριοθετημένη Μνήμη για το Πρόβλημα Παραγωγού- Καταναλωτή

```
producer:
while (true) {
    /* produce item v */
    while ((in + 1) % n == out)
        /* full - do nothing */
    b[in] = v;
    in = (in + 1) % n;
}
consumer:
while (true) {
    while (in == out)
        /* empty - do nothing */;
    w = b[out];
    out = (out + 1) % n;
    /* consume item w */
}
```


Υλοποίηση με Γενικές Σημαφόρους

```
/* program bounded buffer */
semaphore mutual_exclusion=1;
semaphore full_slots=0;
semaphore empty_slots=BUFFERSIZE;

void producer() {
    while (true) {
        produce();
        wait(empty_slots);
        wait(mutual_exclusion);
        append();
        signal(mutual_exclusion);
        signal(full_slots);
    }
}
```

```
void consumer() {
    while (true) {
        wait(full_slots);
        wait(mutual_exclusion);
        take();
        signal(mutual_exclusion);
        signal(empty_slots);
        consume();
    }
}

void main() {
  parbegin(producer, consumer);
}
```

Υλοποίηση Σημαφόρων

- Θεωρούμε
  - Σύστημα ενός επεξεργαστή
- wait() και signal()
  - Αδιαίρετες συναρτήσεις
  - Έχουν μικρή διάρκεια
  - Απενεργοποιούν διακοπές
  - Μικρή επιβάρυνση

```
wait(s) {
    disable interrupts;
    s.count--;
    if (s.count < 0) {
        place process in s.queue;
        block process; }
    enable interrupts;
}
signal(s) {
    disable interrupts;
    s.count++;
    if (s.count <= 0) {
        remove process P from s.queue;
        place P on ready list; }
    enable interrupts;
}
```


Υλοποίηση Σημαφόρων με Υλικό

- Θεωρούμε
  - struct s { int flag, count };
  - Εντολή μηχανής testset()
- wait() και signal()
  - Αδιαίρετες συναρτήσεις
  - Ενεργός αναμονή μέχρι flag=0
  - Θέτουν flag = 1 για εισαγωγή
  - Έχουν σύντομη διάρκεια
  - Μικρή επιβάρυνση

```
wait(s) {
    while(!testset(&s.flag))
        /* s.flag = 1 - do nothing */;
    s.count--;
    if (s.count < 0) {
        place process in s.queue;
        block process; }
    s.flag=0;
}

signal(s) {
    while(!testset(&s.flag))
        /* s.flag = 1 - do nothing */ ;
    s.count++;
    if (s.count <= 0) {
        remove process P from s.queue;
        place P on ready list; }
     s.flag=0;
}
```

Παρατηρητές του Hoare [Hoare 1974]

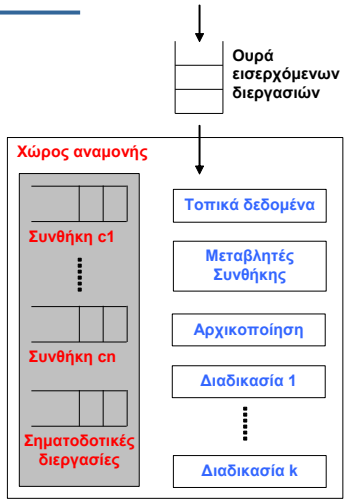
- Ο παρατηρητής είναι ενότητα λογισμικού που αποτελείται από
  - Τοπικά δεδομένα
  - Μια ακολουθία αρχικοποίησης
  - Μία ή περισσότερες διαδικασίες
- Κύρια χαρακτηριστικά
  - Μόνο διαδικασίες παρατηρητή έχουν πρόσβαση σε τοπικές μεταβλητές
  - Μια διεργασία εισέρχεται στον παρατηρητή με κλήση διαδικασίας του
  - Μόνο μια διεργασία εισέρχεται στον παρατηρητή κάθε φορά
    - Οι άλλες διεργασίες που κάλεσαν τον κώδικα παρατηρητή σε αποκλεισμό
    - Περιμένουν να γίνει διαθέσιμος ο παρατηρητής
- Αμοιβαίος αποκλεισμός
  - Εξασφαλίζεται έχοντας μία διεργασία κάθε φορά στον παρατηρητή
  - Προστατεύει τα κοινόχρηστα δεδομένα διατηρώντας τα τοπικά

Συγχρονισμός

- Θεωρήστε μια διεργασία εντός του παρατηρητή
  - Η διεργασία μπαίνει σε αποκλεισμό μέχρι να ισχύσει μια συνθήκη
  - Ο παρατηρητής επιτρέπει μια άλλη διεργασία να εισέλθει
  - Η προηγούμενη διεργασία επανέρχεται όταν ο παρατηρητής γίνει διαθέσιμος και η συνθήκη ικανοποιηθεί
- Μεταβλητές συνθήκης
  - Δηλώνονται εντός του παρατηρητή σαν τοπικά δεδομένα π.χ. int c
  - cwait(c)
    - Βάζει τη διεργασία σε αποκλεισμό στη μεταβλητή συνθήκης c
    - Κάνει τον παρατηρητή διαθέσιμο για μια άλλη διεργασία
  - csignal(c)
    - Επαναφέρει μια διεργασία από αυτές που ήταν σε αποκλεισμό στη c
    - Δεν κάνει τίποτε, αν καμία διεργασία δεν είναι σε αποκλεισμό στη c

Δομή Παρατηρητή

- Συγχρονισμός
  - Δεν υπάρχει μετρητής για κλήσεις csignal() σε μεταβλητές συνθήκης
  - Η csignal() αγνοείται αν δεν περιμένει καμία διεργασία στη μεταβλητή συνθήκης
- csignal() στο τέλος μιας διαδικασίας
  - Η διεργασία εγκαταλείπει τον παρατηρητή
- csignal() στο εσωτερικό διαδικασίας
  - Η διεργασία που καλεί εισέρχεται στην ουρά των σηματοδοτικών διεργασιών



Οριοθετημένη Μνήμη με Παρατηρητή

```
monitor boundedbuffer;
/* monitor local data */
char buffer[BUFFERSIZE];
int nextin, nextout; /*indices*/
int count; /* items in buffer */
int notfull, notempty; /*conditions*/
/* monitor initialization code */
{ nextin = nextout = count = 0; }
/* monitor procedures */
void append(char x) {
    if (count == N) cwait(notfull);
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    csignal(notempty);
}
```

```
void take(char x) {
    if (count == 0) cwait(notempty);
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;
    csignal(notfull);
}
```

```
/*program producer consumer*/
void producer(char x) {
    while (true) { produce(x); append(x); }}
void consumer(char x) {
    while (true) { take(x); consume(x); }}
void main() {
    parbegin(producer, consumer);
}
```

Μειονεκτήματα του Ορισμού του Hoare

- Όταν μια διεργασία καλεί *csignal()* στον παρατηρητή
  - Μπαίνει σε αποκλεισμό (πρώτη αλλαγή διεργασίας)
  - Η διεργασία επανέρχεται όταν γίνει διαθέσιμος ο παρατηρητής (δεύτερη αλλαγή διεργασίας)
  - Οι αλλαγές διεργασίας είναι δαπανηρές
- Όταν μια διεργασία ενεργοποιείται με *csignal()*
  - Η συνθήκη που επανέφερε τη διεργασία πρέπει να ισχύει
  - Η σηματοδοτούμενη διεργασία πρέπει να ξεκινήσει αμέσως
  - Δεν πρέπει να εισέλθει διεργασία που βρίσκεται εκτός παρ/τή
  - Απαιτείται προσεκτική δρομολόγηση για ορθότητα λειτουργίας

Παρατηρητές Mesa [Lampson/Redell 1980]

- Αντικατάσταση του *csignal()* με *cnotify()*
  - Η διεργασία που καλεί *cnotify()* συνεχίζει να τρέχει
  - Η διεργασία στην κεφαλή της ουράς συνθήκης επανέρχεται όταν ο παρατηρητής γίνει διαθέσιμος
- Όταν η σηματοδοτούμενη διεργασία επανέρχεται
  - Η συνθήκη μπορεί να μην ισχύει πλέον
  - Η διεργασία πρέπει να επανεξετάσει τη συνθήκη
  - Αντικαθιστούμε το ***if*** με ***while*** για έλεγχο της συνθήκης
- Βελτίωση
  - Προσθέτουμε στη *cwait()* χρονομετρητή με διάρκεια χρόνου
  - Επαναφέρουμε μια διεργασία όταν λήξει ο χρόνος αναμονής

Οριοθετημένη Μνήμη με Ειδοποίηση

```
void append(char x) {
    while (count == N)
        cwait(notfull);
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    cnotify(notempty);
}
```

```
void take(char x) {
    while (count == 0)
        cwait(notempty);
    x = buffer[nextout];
    nextout = (nextout + 1) %N;
    count--;
    cnotify(notfull);
}
```


Εκπομπή [Lampson/Redell 1980]

- Αντικαθιστούμε τη *csignal()* με *cbroadcast()*
  - Η *cbroadcast()* όταν καλείται επαναφέρει όλες τις διεργασίες που περιμένουν αποκλεισμένες στη συνθήκη
- Χρήσιμη όταν δεν ξέρουμε πόσες διεργασίες να ενεργοποιήσουμε
  - Π.χ. όταν βάζουμε πολλά αντικείμενα στη μνήμη κάθε φορά, αφήνουμε τις διεργασίες να πάρουν όσα χρειάζεται η καθεμία
- Χρήσιμη όταν δεν ξέρουμε ποια διεργασία να ενεργοποιήσουμε
  - Π.χ. όταν περιμένουν πολλαπλές διεργασίες για μνήμη
  - Πρέπει να ενεργοποιηθεί μόνο μία που θα βρει αρκετή μνήμη
  - Τις ξυπνάμε όλες με *cbroadcast()* και τις αφήνουμε να αποφασίσουν
- Ειδοποίηση/εκπομπή
  - Εμφανίστηκαν για πρώτη φορά στη γλώσσα προγραμματισμού *Mesa*

Αμοιβαίος Αποκλεισμός σε Pthreads

- Τα νήματα POSIX χρησιμοποιούν μεταβλητές mutex για να επιτύχουν αμοιβαίο αποκλεισμό

```
pthread_mutex_t mutex;                  /* declaration */
pthread_mutex_init(&mutex, NULL);       /* initialization */

pthread_mutex_lock(&mutex);             /* use */
  /* critical section */
pthread_mutex_unlock(&mutex);

pthread_mutex_destroy(&mutex);          /* deallocation */
```


Μεταβλητές Συνθήκης σε Pthreads

- Ορισμός
  - Σαν τις μεταβλητές συνθήκης σε παρατηρητές Lampson/Redell
    - *phread_cond_wait()/_signal()* αντί για *cwait()/cnotify()*
  - Κάθε μεταβλητή συνθήκης συσχετίζεται με μεταβλητή mutex
- Παράδειγμα

```
pthread_mutex_t mx;                     /* declaration */
pthread_cond_t cv;

pthread_mutex_init(&mx, NULL);          /* initialization */
pthread_cond_init(&cv, NULL);
```

Συγχρονισμός σε Pthreads

```
pthread_mutex_lock(&mx);
  ...
  while (condition)
      pthread_cond_wait(&cv,&mx);
  ...
pthread_mutex_unlock(&mx);
```

```
pthread_mutex_lock(&mx);
  ...
  if (condition)
      pthread_cond_signal(&cv);
  ...
pthread_mutex_unlock(&mx);
```

- *pthread_cond_wait()/_signal()*
  - Πρέπει να περικλείονται σε κλήσεις *pthread_mutex_lock()/_unlock()*
- Συνθήκη πριν την κλήση *pthread_cond_wait()*
  - Πρέπει να επανελέγχεται στο ξύπνημα μέσα σε βρόχο ***while***

Μεταβίβαση Μηνυμάτων

- Μηχανισμός για υποστήριξη
  - Συγχρονισμού: συνεργασία μεταξύ διεργασιών
  - Επικοινωνία: ανταλλαγή πληροφοριών μεταξύ των διεργασίων
- Εφαρμόσιμη σε
  - Μονοπρογραμματιστικά συστήματα
  - Πολυεπεξεργαστές κοινόχρηστης μνήμης
  - Κατανεμημένα συστήματα
- Στοιχειώδης εντολή *send(destination, message)*
  - Στέλνει *message* στη διεργασία *destination*
- Στοιχειώδης εντολή *receive(source, message)*
  - Λαμβάνει *message* από τη διεργασία *source*

Συγχρονισμός

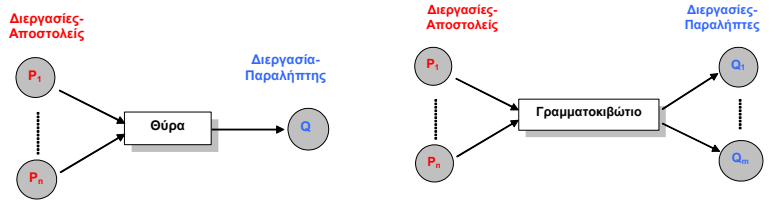
- Αποστολή
  - *Με αποκλεισμό:* ο αποστολέας σε αποκλεισμό μέχρι τη λήψη
  - *Χωρίς αποκλεισμό:* ο αποστολέας συνεχίζει μετά την αποστολή χωρίς να περιμένει για λήψη του μηνύματος (*πιο συνηθισμένη περίπτωση*)
- Λήψη
  - *Με αποκλεισμό:* ο παραλήπτης σε αποκλεισμό μέχρι τη λήψη του μηνύματος (*πιο συνηθισμένη περίπτωση*)
  - *Χωρίς αποκλεισμό:* ο παραλήπτης δεν περιμένει αν δεν ήρθε μήνυμα
  - *Προέλεγχος:* για άφιξη μηνύματος από λίστα αποστολέων
- Περίπτωση 1: Αποστολή/Λήψη με αποκλεισμό (*rendezvous*)
  - Σε αποκλεισμό και οι δύο διεργασίες μέχρι τη μεταφορά μηνύματος
- Περίπτωση 2: Αποστολή χωρίς/Λήψη με αποκλεισμό
  - Αποστολέας συνεχίζει, παραλήπτης σε αποκλεισμό (πιο συνηθισμένη)
- Περίπτωση 3: Αποστολή/Λήψη χωρίς αποκλεισμό
  - Καμία από τις διεργασίες δεν περιμένει

Διευθυνσιοδότηση

- Ορισμός
  - Επιτρέπει τον αποστολέα να καθορίσει τη λαμβάνουσα διεργασία
  - Επιτρέπει τον παραλήπτη να καθορίσει την πηγή του μηνύματος
- Άμεση διευθυνσιοδότηση
  - Ο αποστολέας καθορίζει την ταυτότητα της λαμβάνουσας
  - Ο παραλήπτης καθορίζει την αποστέλλουσα διεργασία ή η εντολή *receive()* επιστρέφει την ταυτότητα του αποστολέα
- Έμμεση διευθυνσιοδότηση
  - Τα μηνύματα στέλνονται σε κοινόχρηστη ουρά (*γραμματοκιβώτιο*)
  - Ο αποστολέας στέλνει στο γραμματοκιβώτιο
  - Ο παραλήπτης λαμβάνει μήνυμα από το γραμματοκιβώτιο
  - Το γραμματοκιβώτιο ανήκει στη διεργασία ή το ΛΣ
  - Η σύνδεση με το γραμματοκιβώτιο είναι στατική ή δυναμική

Σχέσεις στην Έμμεση Διευθυνσιοδότηση

- Αποστολείς και παραλήπτες
  - Μία-προς-μία συνδέει δύο διεργασίες
  - Πολλές-προς-μία συνδέει πολλές διεργασία με μία, το γραμματοκιβώτιο ονομάζεται θύρα (πελάτης/διακομιστής)
  - Μία-προς-πολλές σχετίζει μία δεργασία με πολλές (εκπομπή)
  - Πολλές-προς-πολλές



Μορφή Μηνύματος και Διαχείριση Ουράς

- Μορφή μηνύματος
  - Επικεφαλίδα: ταυτότητα πηγής, ταυτότητα προορισμού, μήκος μηνύματος, τύπος, έλεγχος
  - Σώμα: περιεχόμενα μηνύματος
- Διαχείριση ουράς
  - First-in first-out (FIFO)
  - Με βάση την προτεραιότητα του αποστολέα
  - Με βάση την επιλογή του παραλήπτη

Αμοιβαίος Αποκλεισμός με Μηνύματα

- Θεωρούμε
  - send() χωρίς αποκλεισμό
  - receive() με αποκλεισμό
- Ένα μήνυμα μόνο
  - Η διεργασία που το λαμβάνει επιτρέπεται να εισέλθει

```
/* program mutual exclusion */
const int n = /* # processes */
void main() {
    create_mailbox(mutex);
    send(mutex, null);
    parbegin( P(1), ..., P(n) );
}
```

```
void P(int i) {
    message msg;
    while (true) {
        receive(mutex, msg);
        /* critical section */
        send(mutex, msg);
        /* remainder */
    }
}
```


Οριοθετημένη Μνήμη με Μηνύματα

- Δημιουργία πλήθους μηνυμάτων
  - Ίση με τη χωρητικότητα
- Κάθε μήνυμα *mayproduce*
  - Αντιστοιχεί σε ένα *mayconsume*

```
/* program bounded buffer */
const int
    capacity = /* buffering capacity */,
    null = /* empty message */;

void main() {
    create_mailbox(mayproduce);
    create_mailbox(mayconsume);
    for (int i = 1; i <= capacity; i++)
        send(mayproduce, null);
    parbegin( producer, consumer);
}
```

```
void producer() {
    message pmsg;
    while (true) {
        receive(mayproduce, pmsg);
        pmsg=produce();
        send(mayconsume, pmsg);
    }
}

void consumer() {
    message cmsg;
    while (true) {
        receive (mayconsume, cmsg);
        consume(cmsg);
        send(mayproduce, null);
    }
}
```

Πρόβλημα Αναγνωστών - Γραφέων

- Ορισμός
  - Περιοχή δεδομένων κοινόχρηστη από έναν αριθμό διεργασιών
  - Μερικές διεργασίες μόνο διαβάζουν δεδομένα (*Αναγνώστες*)
  - Μερικές άλλες διεργασίες μόνο γράφουν δεδομένα (*Γραφείς*)
- Συνθήκες
  - Απεριόριστοι αναγνώστες διαβάζουν ταυτόχρονα
  - Μόνο ένας γραφέας μπορεί να γράψει κάθε φορά
  - Αν ένας γραφέας γράφει, κανένας αναγνώστης δε διαβάζει
- Διαφορά από το πρόβλημα του αμοιβαίου αποκλεισμού
  - Εδώ πολλαπλοί αναγνώστες επιτρέπεται να διαβάζουν ταυτόχρονα
- Διαφορά από το πρόβλημα παραγωγού-καταναλωτή
  - Οι αναγνώστες δεν αλλάζουν τα κοινόχρηστα δεδομένα
  - Οι παραγωγοί και καταναλωτές αλλάζουν την κοινόχρηστη δομή

Προτεραιότητες

- Προτεραιότητα στους αναγνώστες
  - Όσο δεν υπάρχουν αναγνώστες οι γραφείς εξυπηρετούνται με τη σειρά άφιξης
  - Έστω ότι καταφθάνει ένας αναγνώστης
  - Οι ακόλουθοι γραφείς περιμένουν όλους τους αναγνώστες που καταφθάνουν
  - Όταν πάψουν να έρχονται αναγνώστες, οι γραφείς μπορούν και πάλι να γράψουν

- Προτεραιότητα στους γραφείς
  - Όσο δεν υπάρχουν γραφείς οι αναγνώστες εξυπηρετούνται κανονικά
  - Έστω ότι καταφθάνει ένας γραφέας
  - Όλοι οι ακόλουθοι αναγνώστες περιμένουν όλους τους γραφείς που καταφθάνουν
  - Όταν πάψουν να έρχονται γραφείς μπορούν πάλι οι αναγνώστες να προχωρήσουν

Προτεραιότητα στους Αναγνώστες

```
/* program readers writers */
int readcount; /* count readers */
semaphore readcount_mutex = 1,
          readwrite_mutex = 1;
void reader() {
    while (true) {
        wait(readcount_mutex);
        readcount++;
        if (readcount == 1)
            wait(readwrite_mutex);
        signal(readcount_mutex);
        READUNIT();
        wait(readcount_mutex);
        readcount--;
        if (readcount == 0)
            signal(readwrite_mutex);
        signal(readcount_mutex);
    }
}
```

```
void writer() {
    while (true) {
        wait(readwrite_mutex);
        WRITEUNIT();
        signal(readwrite_mutex);
    }
}

void main() {
    readcount = 0;
    /* initiate 1 reader+1 writer */
    /* similar solution for multiple */
    parbegin ( reader, writer);
}
```


Προτεραιότητα στους Γραφείς

- readcount_mutex
  - ασφαλής ενημέρωση *readcount*
- writecount_mutex
  - ασφαλής ενημέρωση *writecount*
- read_mutex
  - Εμποδίζει τους αναγνώστες αν υπάρχει ενεργός γραφέας
- read_mutex2
  - Συγκεντρώνει όλους τους αναγνώστες σε αναμονή - 1
  - Δίνει προτεραιότητα στον επόμενο γραφέα
- write_mutex
  - Επιτρέπει έναν γραφέα κάθε φορά

```
/* program readers writers */
int readcount, writecount;
semaphore
        readcount_mutex = 1,
        writecount_mutex = 1,
        read_mutex = 1,
        read_mutex2 = 1,
        write_mutex = 1;

void main() {
    readcount = writecount = 0;
    parbegin(reader, writer);
}
```

Προτεραιότητα στους Γραφείς (συνέχεια)

```
void reader() {
    while (true) {
            wait(read_mutex2);
            wait(read_mutex);
            wait(readcount_mutex);
            readcount++;
            if (readcount == 1)
                    wait(write_mutex);
            signal(readcount_mutex);
            signal(read_mutex);
            signal(read_mutex2);
            READUNIT();
            wait(readcount_mutex);
            readcount--;
            if (readcount == 0);
                    signal(write_mutex);
            signal(readcount_mutex);
    }
}
```

```
void writer() {
    while (true)
            wait(writecount_mutex);
            writecount++;
            if (writecount == 1)
                    wait(read_mutex);
                /* block new readers */
            signal(writecount_mutex);
            wait(write_mutex);
            WRITEUNIT();
            signal(write_mutex);
            wait(writecount_mutex);
            writecount--;
            if (writecount == 0)
                    signal(read_mutex);
                /* release new readers */
            signal(writecount_mutex);
    }
}
```


Διαδιεργασιακή Επικοινωνία στο Unix

- Σωλήνωση
  - FIFO ουρά που γράφεται από μία και διαβάζεται από μία άλλη διεργασία
  - Η *ανώνυμη* περιορίζεται σε διεργασίες που σχετίζονται (π.χ. μέσω fork())
  - Η *επώνυμη* (γνωστή ως FIFO) είναι χρήσιμη σε διεργασίες ξένες μεταξύ τους
- Ουρά μηνυμάτων
  - Παίζει ρόλο γραμματοκιβωτίου
  - Οι διεργασίες στέλνουν και λαμβάνουν μηνύματα καθορισμένου τύπου
- Κοινόχρηστη μνήμη
  - Χώρος μνήμης που απεικονίζεται στην ιδεατή μνήμη πολλών διεργασιών
- Σημαφόρος
  - Γενικευμένη σημαφόρος με επιλογές ενημέρωσης μετρητή και αποκλεισμού ή αφύπνησης διεργασιών
- Σήμα
  - Πληροφορεί μια διεργασία για ένα ασύγχρονο γεγονός
  - Αποθηκεύεται ως μοναδικό δυαδικό ψηφίο στη δομή ελέγχου μιας διεργασίας