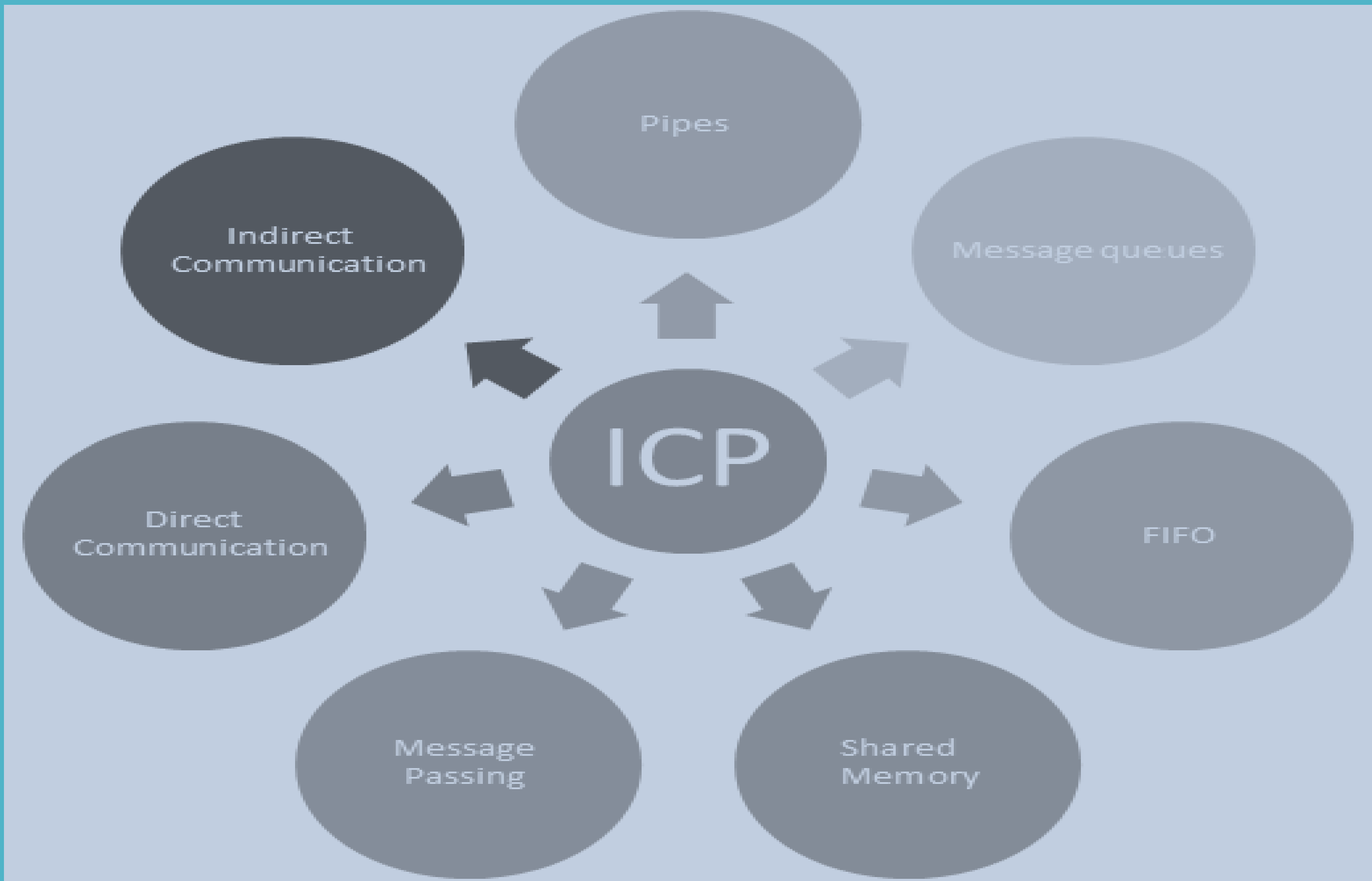
Διαδιεργασιακή επικοινωνία



Διαδιεργασιακή επικοινωνία

Στα μοντέρνα λειτουργικά συστήματα, η διαχείριση πολλαπλών διεργασιών μπορεί να αφορά σε

- Ένα σύστημα απλού επεξεργαστή (πολυπρογραμματισμός)
- Ένα σύστημα πολυεπεξεργαστή (πολυεπεξεργασία)
- Ένα σύστημα κατανεμημένης επεξεργασίας

Η ταυτόχρονη εκτέλεση πολλών διεργασιών οι οποίες μάλιστα μπορούν να αλληλεπιδρούν μεταξύ τους, είναι γνωστή ως ταυτοχρονισμός (concurrency) και σχετίζεται με πολύ σημαντικά ζητήματα σχεδίασης όπως είναι ο διαμοιρασμός πόρων και ο ανταγωνισμός.

Υπάρχουν δύο τρόποι εκκίνησης διεργασιών οι οποίες επικοινωνούν μεταξύ τους:

- Η μία διεργασία ξεκινά τη fork για να ξεκινήσει την άλλη διεργασία (πατέρας - παιδί).
- Η κάθε διεργασία εκτελείται από το δικό της τερματικό (ανεξάρτητες διεργασίες).

Αναγκαία προϋπόθεση για την υποστήριξη του ταυτοχρονισμού είναι ο αμοιβαίος αποκλεισμός, δηλαδή ο αποκλεισμός όλων των υπόλοιπων διεργασιών από μία ροή ενεργειών η οποία εκτελείται σε κάθε χρονική στιγμή από κάποια συγκεκριμένη διεργασία.

Οι μηχανισμοί υποστήριξης ταυτοχρονισμού περιλαμβάνουν μεταξύ άλλων τους σημαφόρους (semaphores), τους παρακολουθητές (watchers) και τη μεταβίβαση μηνυμάτων (message passing).

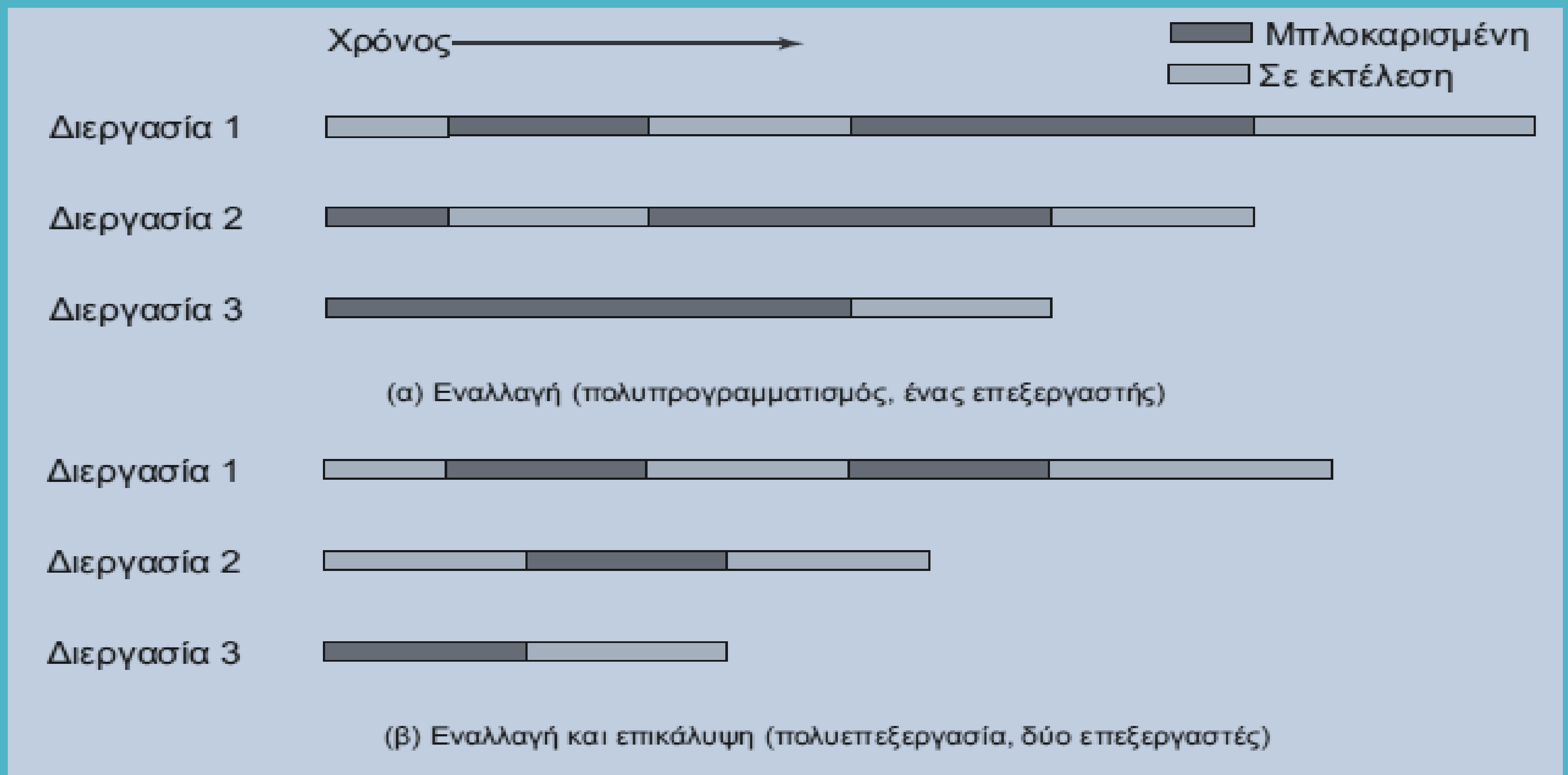
Διαδιεργασιακή επικοινωνία

Οι πιο σημαντικοί όροι που σχετίζονται με τον ταυτοχρονισμό διεργασιών, είναι οι ακόλουθοι:

- Ατομική λειτουργία ή ατομικότητα: ενέργεια που υλοποιείται ως ακολουθία εντολών οι οποίες εμφανίζονται αδιαίρετες. Αυτή η ακολουθία είτε εκτελείται στο σύνολό της είτε δεν εκτελείται καθόλου.
- Κρίσιμο τμήμα: τμήμα κώδικα που σχετίζεται με τη λειτουργία κοινόχρηστου πόρου και δεν πρέπει να εκτελεστεί τη στιγμή που μία άλλη διεργασία εκτελεί ένα αντίστοιχο τμήμα κώδικα.
- Αδιέξοδο: κατάσταση που ανακύπτει όταν δύο ή περισσότερες διεργασίες δεν μπορούν να προχωρήσουν επειδή η καθεμία από αυτές περιμένει την ολοκλήρωση της λειτουργίας κάποιας άλλης.
- Αλληλεξάρτηση: κατάσταση που προκύπτει όταν δύο ή περισσότερες διεργασίες αλλάζουν συνεχώς τις καταστάσεις τους ως αντίδραση σε μεταβολές που πραγματοποιούνται σε άλλες διεργασίες.
- Αμοιβαίος αποκλεισμός: προϋποθέτει πως όταν μία διεργασία βρίσκεται στο κρίσιμο τμήμα της, καμία άλλη διεργασία δεν μπορεί να βρίσκεται στο δικό της κρίσιμο τμήμα (αλγόριθμοι Decker και Peterson).
- Συνθήκη ανταγωνισμού: κατάσταση στην οποία πολλαπλές διεργασίες διαβάσουν ή γράφουν ένα διαμοιραζόμενο αρχείο και το τελικό αποτέλεσμα εξαρτάται από τη σειρά της εκτέλεσής τους.
- Λιμοκτονία: κατάσταση κατά την οποία μία διεργασία αγνοείται επ’ αόριστον από τον επεξεργαστή.

Διαδιεργασιακή επικοινωνία

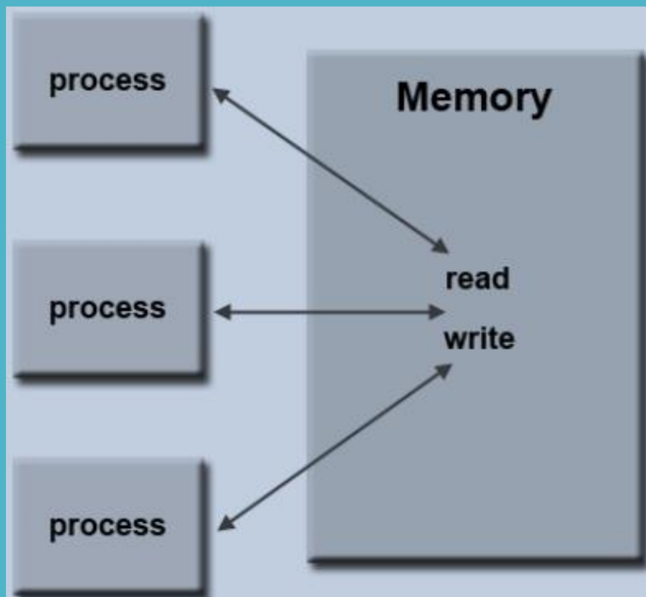
Σε συστήματα απλού επεξεργαστή οι διεργασίες εναλλάσσονται χρονικά έτσι ώστε να δίδεται η εντύπωση ταυτόχρονης εκτέλεσης. Στα συστήματα των πολυεπεξεργαστών εκτός από χρονική εναλλαγή υποστηρίζεται και η χρονική επικάλυψη.

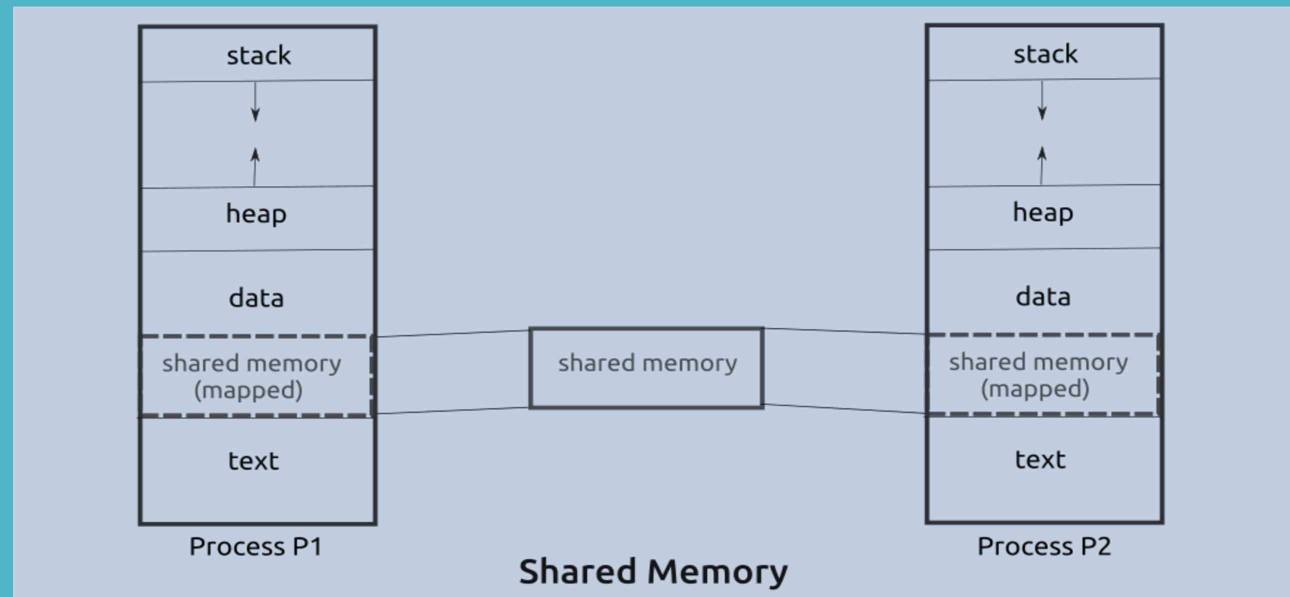


Διαδιεργασιακή επικοινωνία

Στην επιστήμη της πληροφορικής χρησιμοποιούμε τον όρο κοινόχρηστος πόρος για να περιγράψουμε κάθε είδος πόρου (συνήθως περιοχές μνήμης και αρχεία) τα οποία μπορούν να προσπελαστούν από πολλές διεργασίες ταυτόχρονα.

Κοινόχρηστη μνήμη (shared memory, shm) --> δημιουργείται από τον πυρήνα του λειτουργικού ο οποίος την αντιστοιχεί σε κάποια περιοχή του τμήματος δεδομένων της κάθε διεργασίας.





Κοινόχρηστο αρχείο --> επιτρέπει την ταυτόχρονη προσπέλασή του από πολλές διεργασίες για διαδικασίες ανάγνωσης ή εγγραφής.

Διαδιεργασιακή επικοινωνία

Η χρήση κοινόχρηστων πόρων ενδέχεται να οδηγήσει σε προβλήματα απροσδιόριστης έκβασης και συνοχής δεδομένων. Τι γίνεται όταν δύο διεργασίες προσπελαύνουν ταυτόχρονα την ίδια θέση μνήμης η μία για ανάγνωση και η άλλη για εγγραφή?

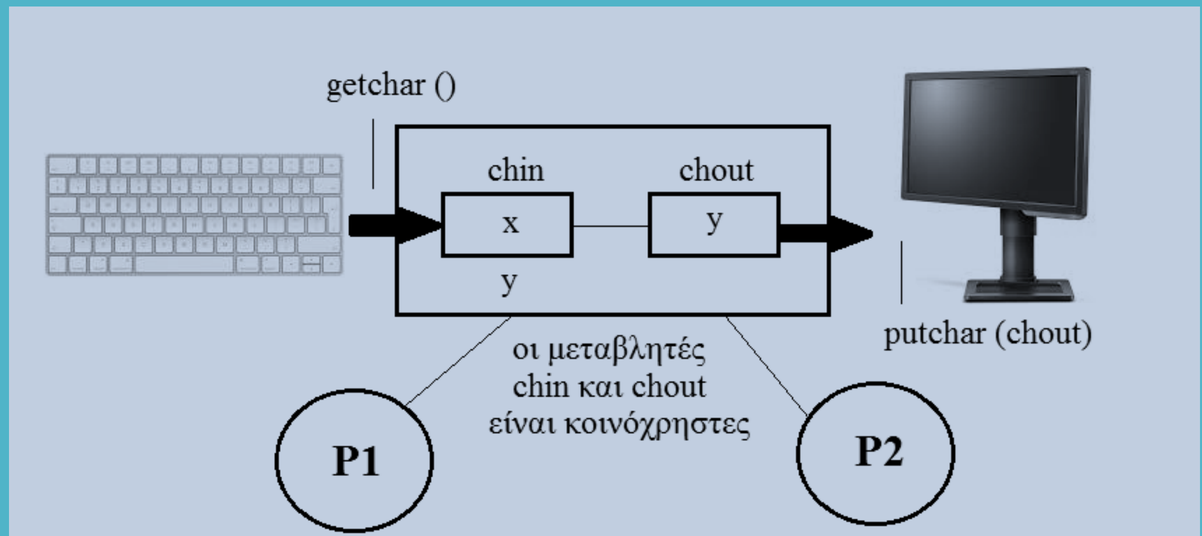
Έστω δύο διεργασίες P1 και P2 που καλούν την εντολή echo.
Οι μεταβλητές chin και chout είναι κοινόχρηστες στις P1 και P2.

```
void echo(){
   chin = getchar();
   chout = chin;
   putchar(chout); }
```

Η P1 καλεί την echo και διακόπτεται όταν εκτελείται η εντολή chin = getchar () με chin = x.

Η P2 ενεργοποιείται και καλεί την echo η οποία ολοκληρώνεται με την εκχώρηση chout = chin = y.

Όταν ενεργοποιείται η P2, η chin έχει την τιμή y η οποία εκτυπώνεται για δεύτερη φορά, ενώ η τιμή x έχει χαθεί οριστικά !!



Διαδιεργασιακή επικοινωνία

ΛΥΣΗ → Επιτρέπουμε σε ΜΙΑ ΜΟΝΟ διεργασία να καλέσει τη συνάρτηση echo. Στην περίπτωση αυτή:

- Η διεργασία P1 εκτελεί την πρώτη εντολή της echo και στη συνέχεια αναστέλλεται.
- Η διεργασία P2 ενεργοποιείται και καλεί την echo. Ωστόσο, επειδή η διεργασία P1 βρίσκεται μέσα στην echo (παρά το γεγονός πως είναι ανεσταλμένη) η διεργασία P2 μπλοκάρεται.
- Η διεργασία P1 συνεχίζει τη λειτουργία της και ολοκληρώνει την κλήση της echo.
- Η διεργασία P2 ξεμπλοκάρει και καλεί με τη σειρά της την echo την οποία και εκτελεί.

Συνθήκη ανταγωνισμού → το τελικό αποτέλεσμα εξαρτάται από τη σειρά εκτέλεσης.

- Έστω διεργασίες P1 και P2 και κοινόχρηστες μεταβλητές b=1 και c=2.
- Η P1 εκτελεί την εντολή b = b + c και η P2 την εντολή c = b + c.
- Εάν εκτελεστεί πρώτη η P1 και μετά η P2 οι νέες τιμές θα είναι b = 3 και c = 5.
- Εάν εκτελεστεί πρώτη η P2 και μετά η P1 οι νέες τιμές θα είναι c = 3 και b = 4.

Η ΑΝΑΓΚΑΙΟΤΗΤΑ ΑΜΟΙΒΑΙΟΥ ΑΠΟΚΛΕΙΣΜΟΥ ΕΠΙΦΕΡΕΙ ΑΔΙΕΞΟΔΑ ΚΑΙ ΛΙΜΟΚΤΟΝΙΕΣ !!

Διαδιεργασιακή επικοινωνία

Σημαφόροι και μεταβλητές αμοιβαίου αποκλεισμού (Dijkstra, 1965)

Θεμελιώδης κανόνας → Δύο ή περισσότερες διεργασίες μπορούν να συνεργάζονται μεταξύ τους μέσω απλών μηνυμάτων με τέτοιο τρόπο ώστε μία διεργασία να εξαναγκάζεται να σταματήσει σε μία συγκεκριμένη θέση όταν λάβει ένα τέτοιο σήμα.

Η σηματοδοσία πραγματοποιείται με τη βοήθεια ειδικών μεταβλητών που ονομάζονται σημαφόροι (semaphores, sem) και οι οποίες αρχικοποιούνται σε μη αρνητικές ακέραιες τιμές.

Πώς χρησιμοποιούνται οι σημαφόροι?

Η συνάρτηση semWait (s) μειώνει την τιμή του σημαφόρου s κατά μία μονάδα. Αν η τιμή του s γίνει αρνητική, τότε μπλοκάρεται η διεργασία που κάλεσε την semWait, διαφορετικά συνεχίζει κανονικά.

Η συνάρτηση semSignal (s) αυξάνει την τιμή του σημαφόρου s κατά μία μονάδα. Αν η τιμή του s μικρότερη ή ίση με το μηδέν, τότε ξεμπλοκάρεται μία διεργασία (εάν υπάρχει) που έχει μπλοκαριστεί από μία συνάρτηση semWait.

Διαδιεργασιακή επικοινωνία

Σημαφόροι και μεταβλητές αμοιβαίου αποκλεισμού (Dijkstra, 1965)

```
struct semaphore {
      int count;
      queueType queue;
};
void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0) {
            /* τοποθέτηση της διεργασίας αυτής στην s.queue */;
            /* μπλοκάρισμα της διεργασίας αυτής */;
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <= 0) {
            /* αφαίρεση μίας διεργασίας P από την s.queue */;
            /* τοποθέτηση της διεργασίας P στην έτοιμη λίστα */;
      }
}
```

Κανείς δεν μπορεί να γνωρίζει αν μία διεργασία θα μπλοκαριστεί η όχι, πριν αυτή μειώσει έναν σημαφόρο.

Εάν η αύξηση του σημαφόρου ενεργοποιήσει κάποια άλλη διεργασία, αμφότερες εκτελούνται ταυτόχρονα.

Όταν αποστέλλεται σήμα σε σημαφόρο κανείς δεν γνωρίζει εάν απαραίτητα κάποια διεργασία βρίσκεται σε αναμονή και κατά συνέπεια το πλήθος των διεργασιών που ενεργοποιούνται μπορεί να είναι 0 ή 1.

Διαδιεργασιακή επικοινωνία

Σημαφόροι και μεταβλητές αμοιβαίου αποκλεισμού (Dijkstra, 1965)

Αρχικά η τιμή του σημαφόρου (γενικός σημαφόρος ή σημαφόρος μέτρησης) είναι θετική ή μηδενική.

- Θετική τιμή → ίση με το πλήθος των διεργασιών που μπορούν να εκδώσουν αίτημα αναμονής και αμέσως μετά να συνεχίσουν να εκτελούνται.
- Μηδενική τιμή → το λειτουργικό μπλοκάρει την επόμενη διεργασία που θα ζητήσει αίτημα αναμονής και η τιμή του σημαφόρου γίνεται αρνητική. Κάθε επακόλουθη αναμονή καθιστά την τιμή του σημαφόρου ακόμη πιο αρνητική – στην περίπτωση αυτή η αρνητική τιμή εκφράζει το πλήθος των διεργασιών που αναμένουν να ξεμλοκαριστούν.

Δυαδικοί σημαφόροι (binary semaphores) – παίρνουν μόνο τις τιμές 0 ή 1

Η λειτουργία τους ελέγχεται από τη συνάρτηση semWaitB. Εάν είναι s = 0 μπλοκάρεται η διεργασία που καλεί την semWaitB. Αν είναι s = 1 τότε θέτουμε s = 0 και η διεργασία συνεχίζει να εκτελείται.

Η συνάρτηση semSignalB ελέγχει αν υπάρχουν μποκαρισμένες διεργασίες σε αυτόν τον σημαφόρο – δηλαδή να έχουν s = 0. Εάν υπάρχουν, κάποια από αυτές ξεμπλοκάρεται (με πολιτική FIFO για τους ισχυρούς σημαφόρους και χωρίς πολιτική για τους ασθενείς σημαφόρους) , εάν όχι θέτουμε s = 1.

Οι δυαδικοί σημαφόροι έχουν την ίδια εκφραστική δύναμη με τους γενικούς σημαφόρους.

Διαδιεργασιακή επικοινωνία

Κλειδαριές αμοιβαίου αποκλεισμού

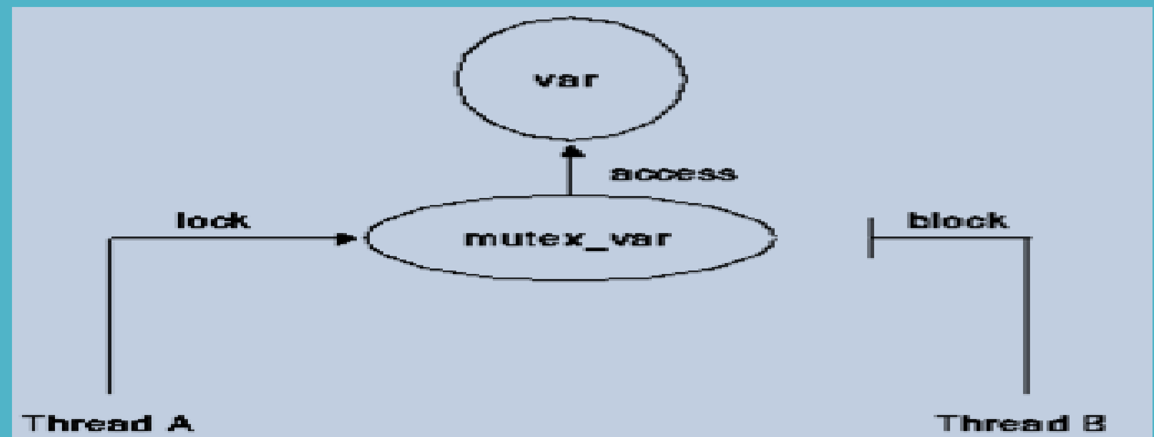
Οι κλειδαριές αμοιβαίου αποκλεισμού (mutual exclusion locks, mutex) είναι παρόμοιες με τους δυαδικούς σημαφόρους που χρησιμοποιούνται για τη δέσμευση και την αποδέσμευση αντικειμένων.

Είναι παρόμοιες με τους δυαδικούς σημαφόρους που χρησιμοποιούνται για τη δέσμευση και την αποδέσμευση αντικειμένων.

Όταν αποκτώνται δεδομένα που δεν μπορούν να αποτελέσουν αντικείμενο διαμοιρασμού ή εκκινείται επεξεργασία η οποία δεν μπορεί να εκτελεστεί κάπου αλλού στο σύστημα, τότε η mutex τίθεται σε κατάσταση κλειδώματος, λαμβάνοντας την τιμή 0.

Όταν δεν απαιτούνται πλέον αυτά τα δεδομένα ή όταν έχει τερματιστεί η παραπάνω επεξεργασία, τότε η mutex τίθεται σε κατάσταση ξεκλειδώματος, λαμβάνοντας την τιμή 1.

Η διαφορά ανάμεσα σε μία mutex και σε έναν δυαδικό σημαφόρο, είναι πως η διεργασία που κλειδώνει τη mutex πρέπει να είναι και αυτή που θα την ξεκλειδώσει, ενώ ένας δυαδικός σημαφόρος μπορεί να κλειδωθεί από μία διεργασία και να ξεκλειδωθεί από μία άλλη.



Διαδιεργασιακή επικοινωνία

Στο λειτουργικό σύστημα Linux η επικοινωνία μεταξύ των διεργασιών μπορεί να πραγματοποιηθεί χρησιμοποιώντας τις επόμενες προσεγγίσεις:

- Αγωγοί (pipes) ανώνυμοι και επώνυμοι.
- Σήματα (signals) που ανταλλάσσονται ανάμεσα στις διεργασίες και στον πυρήνα.
- Υποδοχείς (sockets) που επιτρέπουν την επικοινωνία διεργασιών οι οποίες εκτελούνται στον ίδιο ή σε διαφορετικούς υπολογιστές.
- **Ουρές μηνυμάτων (message queues, msg).**
- **Σημαφόρους (semaphores, sem) & κοινόχρηστα αρχεία (shared files).**
- **Κοινόχρηστες περιοχές μνήμης (shared memory, shm).**

Εκτός από τις παραπάνω υπάρχουν και άλλες τεχνικές όπως είναι η μεταβίβαση μηνυμάτων (message passing) που σχετίζεται με τα παράλληλα και κατανεμημένα συστήματα.

Διαδιεργασιακή επικοινωνία

Η διαδιεργασιακή επικοινωνία με δομές FIFO (επώνυμους αγωγούς) χαρακτηρίζεται από τα ακόλουθα πλεονεκτήματα και μειονεκτήματα:

Πλεονεκτήματα FIFO:

- Είναι εύκολοι στην κατανόηση και στη χρήση τους.
- Είναι διαθέσιμοι σε όλες τις εκδοχές και εκδόσεις του λειτουργικού συστήματος UNIX.
- Είναι αρκετά αποδοτικοί.
- Λειτουργούν καλά τόσο με ρεύματα δεδομένων όσο και με απλά μηνύματα.

Μειονεκτήματα FIFO:

- Μία δομή FIFO δεν μπορεί να έχει πολλούς αναγνώστες.
- Απαιτείται προσωρινή αποθήκευση η οποία είναι αρκετά δαπανηρή.
- Αν το μέγεθος του μηνύματος είναι πολύ μεγάλο, ο συγγραφέας μπορεί να μπλοκάρει.

Το πρόβλημα παραγωγού καταναλωτή → θα πρέπει να διασφαλιστεί πως ο παραγωγός (συγγραφέας) δεν θα πρέπει να προσθέσει δεδομένα στον αγωγό εάν αυτός είναι γεμάτος και πως ο καταναλωτής (αναγνώστης) δεν θα προσπαθήσει να διαβάσει δεδομένα από έναν άδειο αγωγό.

Διαδιεργασιακή επικοινωνία

Η διαδιεργασιακή επικοινωνία με σήματα (signals) χαρακτηρίζεται από τα ακόλουθα πλεονεκτήματα και μειονεκτήματα:

Πλεονεκτήματα σημάτων

- Επιτρέπουν ασύγχρονη λειτουργία (το σήμα στέλνεται όταν απαιτείται κάτι τέτοιο).
- Ως αποτέλεσμα, οδηγούν σε μεγαλύτερη απόδοση της εφαρμογής.
- Προσφέρουν εύκολο χειρισμό διεργασιών μέσω της συνάρτησης αποστολής σημάτων kill.

Μειονεκτήματα σημάτων

- Οδηγούν σε καταστάσεις ανταγωνισμού υπό την έννοια πως ίσως η έκβαση μιας διεργασίας να εξαρτάται από τη σειρά αποστολής των σημάτων.
- Οι εφαρμογές που χρησιμοποιούν σήματα είναι δύσκολες στην αποσφαλμάτωση.
- Ανακύπτουν προβληματικές καταστάσεις όταν οι διεργασίες παραλαμβάνουν ένα σήμα τη στιγμή που έκτελούν τη συνάρτηση του χειρισμού τους.
- Οι διεργασίες δεν χαρακτηρίζονται από μεγάλο βαθμό απομόνωσης.

Διαδιεργασιακή επικοινωνία

Η διαδιεργασιακή επικοινωνία με υποδοχείς (sockets) χαρακτηρίζεται από τα ακόλουθα πλεονεκτήματα και μειονεκτήματα:

Πλεονεκτήματα υποδοχών

- Υποστηρίζουν πλήρως αμφίδρομη επικοινωνία, σε αντίθεση με τους αγωγούς που επιτρέπουν τη μετακίνηση δεδομένων μόνο προς τη μία κατεύθυνση. Ωστόσο είναι πιο δύσχρηστα από τους αγωγούς στο να διασφαλίσουν την χωρίς σφάλματα διακίνηση της διακινούμενης πληροφορίας.
- Η διαδικασία αποσφαλμάτωσης είναι σχετικά απλή ενώ διασφαλίζεται η φορητότητα.
- Επιτρέπουν την εύκολη υλοποίηση εφαρμογών client – server.

Μειονεκτήματα υποδοχών

- Η επιβάρυνση του συστήματος είναι μεγαλύτερη και όταν είναι γνωστό πως η εφαρμογή θα εκτελεστεί σε έναν απλό υπολογιστή είναι προτιμότερη η λύση της κοινόχρηστης μνήμης.
- Απαιτείται ιδιαίτερη προσοχή για τη διασφάλιση της εμπιστευτικότητας των πληροφοριών ειδικά σε περιπτώσεις στις οποίες η εφαρμογή μπορεί να προσπελαστεί και από τρίτους.

Διαδιεργασιακή επικοινωνία

Η διαδιεργασιακή επικοινωνία (Interprocess Communication, IPC) που στηρίζεται στη χρήση σημαφόρων, ουρών μηνυμάτων και κοινόχρηστης μνήμης, πραγματοποιείται με τη βοήθεια δύο διαφορετικών ομάδων κλήσεων συναρτήσεων:

System V IPC και POSIX IPC

Στο υποσύστημα IPC του System V υπάρχουν και τα τρία παραπάνω είδη αντικειμένων, δηλαδή

- Ουρές μηνυμάτων (Message Queues, msg)
- Σημαφόροι (Semaphores, sem)
- Κοινόχρηστη μνήμη (Shared Memory, shm)

Για τη διαχείριση αυτών των αντικειμένων υπάρχουν δύο οικογένειες συναρτήσεων με ονόματα Xget και Xctl όπου το X μπορεί να είναι ένα από τα msg, sem, shm. Υπάρχουν λοιπόν έξι συναρτήσεις, οι

msgget, msgctl, shmget, shmctl, semget, semctl

ενώ επιπλέον υπάρχουν άλλες πέντε συναρτήσεις με ονόματα

msgsnd, msgrcv, shmat, shmdt, semop

Διαδιεργασιακή επικοινωνία

Ιδιότητες των αντικειμένων του System V IPC

- Υφίστανται μόνο σε ένα απλό υπολογιστή.
- Η διάρκεια ζωής τους είναι η ίδια με αυτή του πυρήνα και κατά συνέπεια καταστρέφονται κατά την επανεκκίνηση του υπολογιστή.
- Προσπελαύνονται με τη βοήθεια ενός ακέραιου αναγνωριστικού το οποίο είναι σταθερό σε όλη τη διάρκεια της ζωής του αντικειμένου. Όποια διεργασία γνωρίζει αυτό το αναγνωριστικό μπορεί να προσπελάσει το αντικείμενο χωρίς να χρειαστεί να το ανοίξει.
- Η τιμή του αναγνωριστικού για το κάθε αντικείμενο είναι διαφορετική σε κάθε επανεκκίνηση ενώ η διαδικασία λήψης του αναγνωριστικού διευκολύνεται από τη χρήση ενός μόνιμου κλειδιού.
- Δεν υπάρχουν i-nodes ή pathnames και κατά συνέπεια δεν μπορούν να χρησιμοποιηθούν οι παραδοσιακές κλήσεις συστήματος όπως είναι η unlink, stat, read και write.
- Το κάθε αντικείμενο χαρακτηρίζεται από άδειες πρόσβασης για ανάγνωση, εγγραφή και εκτέλεση παρόμοιες με αυτές των αρχείων αν και το bit εκτέλεσης δεν χρησιμοποιείται, αφού τα αντικείμενα αυτά δεν εκτελούνται.
- Το κάθε αντικείμενο διαθέτει επίσης αναγνωριστικά για τον κάτοχο και την ομάδα του κατόχου τα οποία προσπελαύνονται και τροποποιούνται με τις συναρτήσεις Xctl.

Διαδιεργασιακή επικοινωνία

POSIX IPC

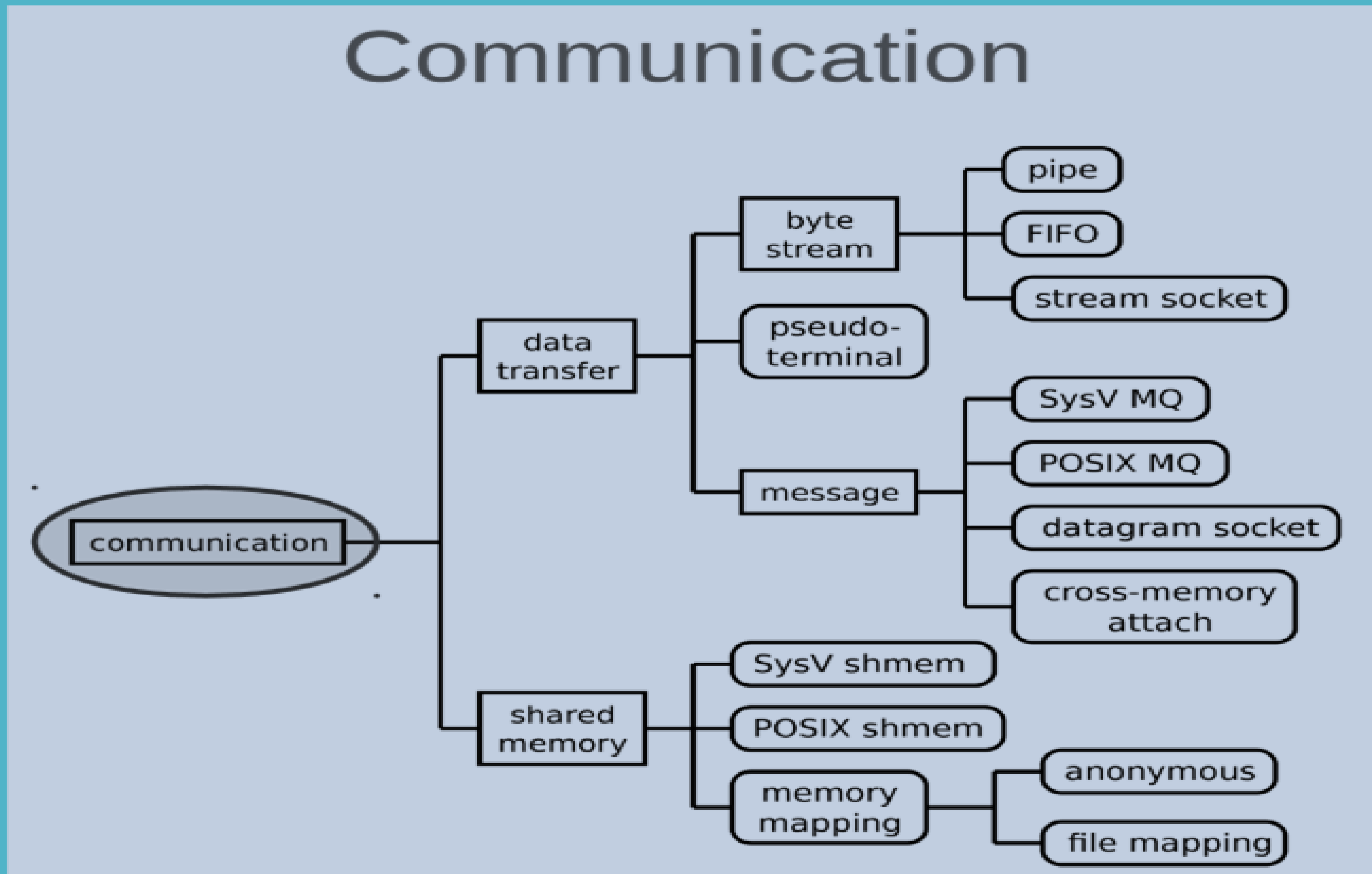
- Στο POSIX τη θέση των κλειδιών παίρνουν κατάλληλα ορισμένα αλφαριθμητικά.
- Τα ονόματα των αντικειμένων ακολουθούν τους ίδιους κανόνες με τα ονόματα των διαδρομών (εάν το όνομα ξεκινά με κάθετο, όλες οι αναφορές σε αυτό σχετίζονται με το ίδιο αντικείμενο).
- Τα αντικείμενα του POSIX IPC (όπως και του System V IPC) δεν είναι αρχεία.

Στο λειτουργικό σύστημα Linux, το πλήρες σύστημα IPC περιλαμβάνει αντικείμενα για πραγματοποίηση δύο θεμελιωδών λειτουργιών

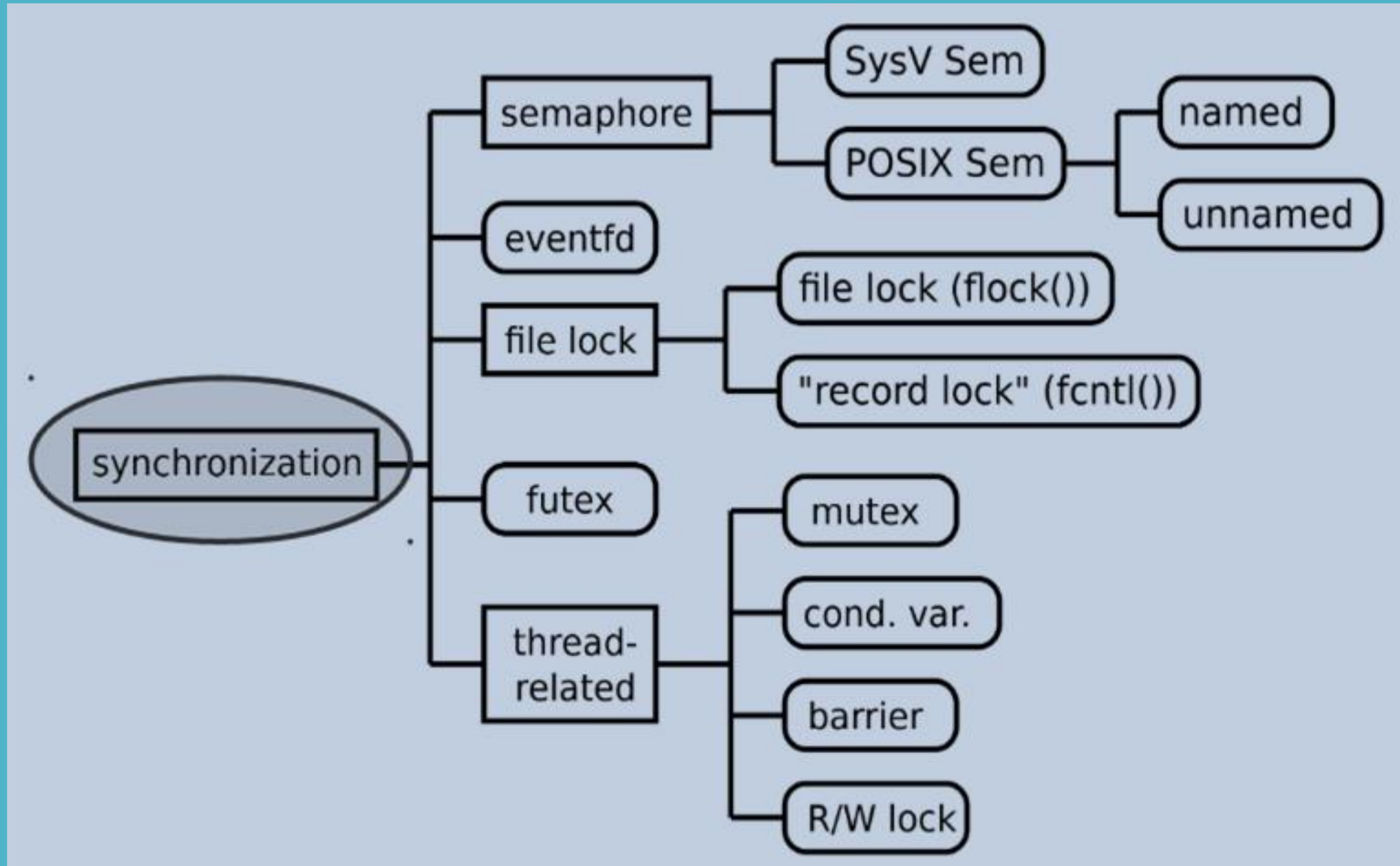
Επικοινωνία & Συγχρονισμός

Τα πιο σημαντικά αντικείμενα της πρώτης κατηγορίας είναι η κοινόχρηστη μνήμη, οι αγωγοί, οι υποδοχείς και οι ουρές μηνυμάτων, ενώ στη δεύτερη κατηγορία ανήκουν οι σημαφόροι και οι μεταβλητές αμοιβαίου αποκλεισμού.

Διαδιεργασιακή επικοινωνία



Διαδιεργασιακή επικοινωνία



Διαδιεργασιακή επικοινωνία

Κοινόχρηστα αρχεία

Δύο διεργασίες ανοίγουν ταυτόχρονα το ίδιο αρχείο, η μία για εγγραφή και η άλλη για ανάγνωση.

Εάν η διεργασία - συγγραφέας προσπελάσει το αρχείο ακριβώς την ίδια στιγμή με τη διεργασία – αναγνώστης, το αποτέλεσμα που θα προκύψει μπορεί να είναι απροσδιόριστο.

Προκειμένου να αποφύγουμε τέτοια προβλήματα, η διεργασία – συγγραφέας θα πρέπει να κλειδώσει το αρχείο πριν την έναρξη της διαδικασίας εγγραφής δεδομένων σε αυτό, έτσι ώστε κατά τη χρονική διάρκεια της εγγραφής να μην είναι δυνατή η προσπέλαση του αρχείου από άλλη διεργασία για ανάγνωση ή εγγραφή. Όταν η εγγραφή ολοκληρωθεί, το αρχείο ξεκλειδώνει και μπορεί πλέον να χρησιμοποιηθεί και από άλλες διεργασίες **(exclusive lock)**.

Από την άλλη πλευρά, η ανάγνωση δεν τροποποιεί τα δεδομένα του αρχείου και γενικά είναι επιτρεπτή η ταυτόχρονη ανάγνωση του αρχείου από πολλές διεργασίες. Για μία ακόμη φορά απαιτείται κλείδωμα του αρχείου αλλά αυτή τη φορά δεν είναι αποκλειστικό αλλά κοινόχρηστο **(shared lock)** έτσι ώστε η ανάγνωση να μπορεί να πραγματοποιηθεί από πολλές διεργασίες. Ωστόσο για όσο χρονικό διάστημα ένας αναγνώστης έχει ένα κοινόχρηστο κλείδωμα, η εγγραφή στο αρχείο δεν επιτρέπεται.

Η διαχείριση των shared και exclusive locks γίνεται από τη συνάρτηση **fcntl** .

Διαδιεργασιακή επικοινωνία

Κοινόχρηστα αρχεία (lock & unclock)

Το πρωτότυπο της συνάρτησης fcntl (unistd.h και fcntl.h) έχει τη μορφή

int fcntl (int fd, int cmd, [opt arg]);

Η συνάρτηση επιτρέπει το χειρισμό ενός ανοικτού αρχείου που περιγράφεται από το πρώτο όρισμα fd, σύμφωνα με το δεύτερο όρισμα cmd ενώ η συμπεριφορά της καθορίζεται από το τρίτο όρισμα που είναι μία δομή τύπου flock της μορφής

```
struct flock {
  short  l_type;    /* F_RDLCK, F_WRLCK, or F_UNLCK */
  short  l_whence;  /* SEEK_SET, SEEK_CUR, or SEEK_END */
  off_t  l_start;   /* offset in bytes, relative to l_whence */
  off_t  l_len;     /* length, in bytes; 0 means lock to EOF */
  pid_t  l_pid;     /* returned with F_GETLK */
};
```

Σχετικά με το πρώτο όρισμα αυτό μπορεί να λάβει κάποια από τις τιμές F_SETLK που ενεργοποιεί ένα κλείδωμα όταν το πεδίο l_type έχει μία από τις τιμές F_RDLCK ή F_WRLCK) ή καταργεί ένα κλείδωμα όταν το πεδίο l_type έχει την τιμή F_UNLCK για τα bytes του αρχείου που ορίζονται από τα πεδία l_start, l_len και l_whence της δομής flock).

Διαδιεργασιακή επικοινωνία

Παράδειγμα χρήσης κοινόχρηστου αρχείου

```
int main() {
  struct flock lock;
  lock.l_type = F_WRLCK;    /* read/write (exclusive versus shared) lock */
  lock.l_whence = SEEK_SET; /* base for seek offsets */
  lock.l_start = 0;         /* 1st byte in file */
  lock.l_len = 0;           /* 0 here means 'until EOF' */
  lock.l_pid = getpid();    /* process id */

  int fd; /* file descriptor to identify a file within a process */
  if ((fd = open(FileName, O_RDWR | O_CREAT, 0666)) < 0)  {
       perror("open failed..."); exit(-1); }
  if (fcntl(fd, F_SETLK, &lock) < 0) {
       perror("fcntl failed to get lock..."); exit(-2); }
  else {
    write(fd, DataString, strlen(DataString)); /* populate data file */
    fprintf(stderr, "Process %d has written to data file...\n", lock.l_pid); }

  /* Now release the lock explicitly. */
  lock.l_type = F_UNLCK;
  if (fcntl(fd, F_SETLK, &lock) < 0) {
       perror("explicit unlocking failed..."); exit(-1); }

  close(fd);
  return 0; }
```

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define FileName "data.dat"
#define DataString "Now is the winter of our discontent\n"
```

Η διεργασία – συγγραφέας ανοίγει το αρχείο data.dat υπό συνθήκες exclusive lock και αποθηκεύει σε αυτό το περιεχόμενο μιας συμβολοσειράς.

lock.l_type = F_WRLCK
exclusive lock

Διαδιεργασιακή επικοινωνία

Παράδειγμα χρήσης κοινόχρηστου αρχείου

```
int main() {
  struct flock lock;
  lock.l_type = F_WRLCK;    /* read/write (exclusive) lock */
  lock.l_whence = SEEK_SET; /* base for seek offsets */
  lock.l_start = 0;         /* 1st byte in file */
  lock.l_len = 0;           /* 0 here means 'until EOF' */
  lock.l_pid = getpid();    /* process id */

  int fd; /* file descriptor to identify a file within a process */
  if ((fd = open(FileName, O_RDONLY)) < 0) {
      perror("open to read failed..."); exit(-1); }

  /* If the file is write-locked, we can't continue. */
  fcntl(fd, F_GETLK, &lock); /* sets lock.l_type to F_UNLCK if no write lock */
  if (lock.l_type != F_UNLCK) {
      perror("file is still write locked..."); exit(-1); }

  lock.l_type = F_RDLCK; /* prevents any writing during the reading */
  if (fcntl(fd, F_SETLK, &lock) < 0) {
      perror("can't get a read-only lock..."); exit(-1); }

  /* Read the bytes (they happen to be ASCII codes) one at a time. */
  int c; /* buffer for read bytes */
  while (read(fd, &c, 1) > 0)    /* 0 signals EOF */
    write(STDOUT_FILENO, &c, 1); /* write one byte to the standard output */

  /* Release the lock explicitly. */
  lock.l_type = F_UNLCK;
  if (fcntl(fd, F_SETLK, &lock) < 0) {
      perror("explicit unlocking failed..."); exit(-1); }

  close(fd);
  return 0; }
```

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

#define FileName "data.dat"
```

Η διεργασία – αναγνώστης ανοίγει το αρχείο data.dat υπό συνθήκες shared lock και διαβάζει από αυτό το περιεχόμενο της συμβολοσειράς.

lock.l_type = F_RDLCK

shared lock

Διαδιεργασιακή επικοινωνία

Παράδειγμα χρήσης κοινόχρηστου αρχείου

```
amarg@amarg-vbox:~$ ./producer
Process 3249 has written to data file...
amarg@amarg-vbox:~$ ./consumer
Now is the winter of our discontent
```

Το κλείδωμα του αρχείου αναιρείται με δύο τρόπους: είτε άμεσα καλώντας την fnctl

```c
/* Release the lock explicitly. */
lock.l_type = F_UNLCK;
if (fcntl(fd, F_SETLK, &lock) < 0) {
     perror("explicit unlocking failed..."); exit(-1); }
```

είτε έμμεσα, απλά κλείνοντας το αρχείο, αφού προφανώς εάν μία διεργασία κλείσει ένα αρχείο αυτό σημαίνει πως δεν το χρειάζεται πλέον, οπότε δεν έχει λόγο να παραμείνει κλειδωμένο.

Η χρήση του flag F_SETLK στην fcntl

```c
if (fcntl(fd, F_SETLK, &lock) < 0) {
     perror("fcntl failed to get lock..."); exit(-2); }
```

προκαλεί την άμεση επιστροφή της συνάρτησης είτε καταφέρει να κλειδώσει το αρχείο είτε όχι, ενώ εάν χρησιμοποιήσουμε το flag F_SETLKW η συνάρτηση μπλοκάρει μέχρι να κλειδωθεί το αρχείο.

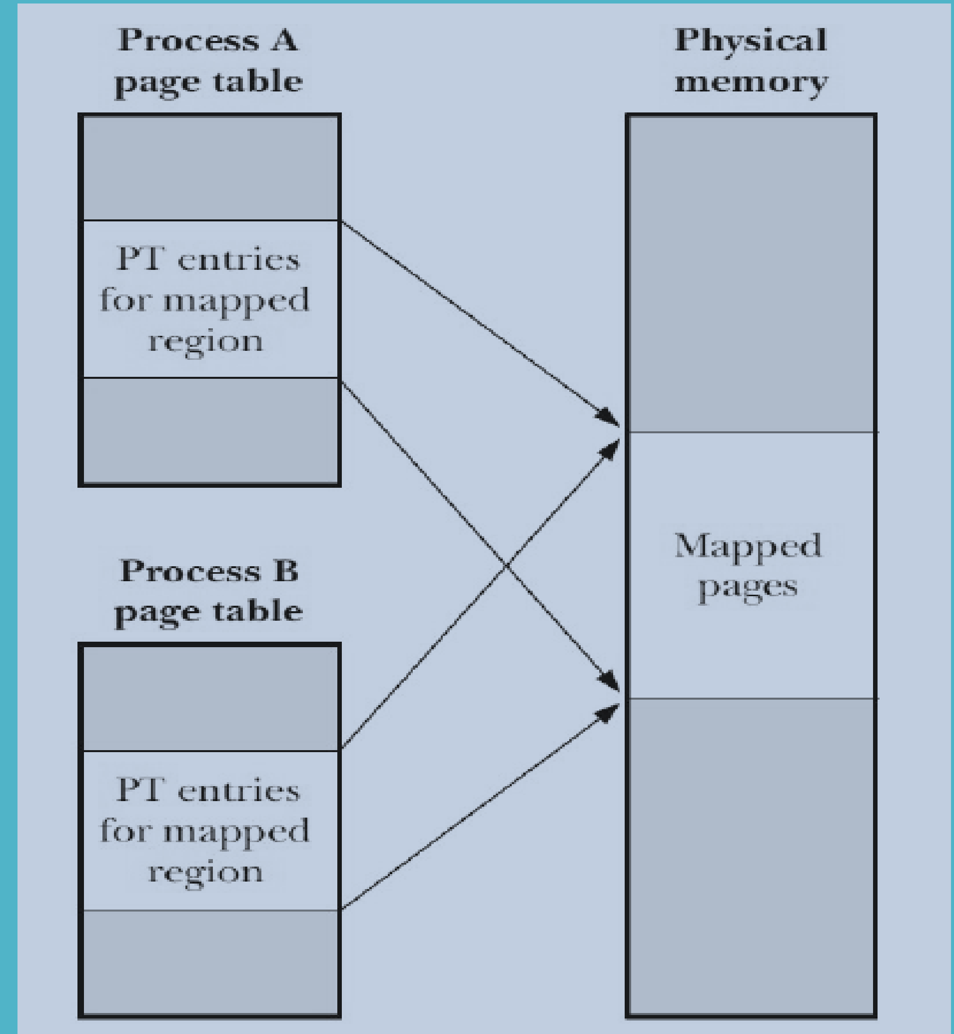
Διαδιεργασιακή επικοινωνία

Κοινόχρηστη μνήμη

Στην προσέγγιση της κοινόχρηστης μνήμης, οι διεργασίες διαμοιράζονται τις ίδιες σελίδες φυσικής μνήμης και η επικοινωνία μεταξύ τους πραγματοποιείται γράφοντας δεδομένα σε αυτή την περιοχή.

Αυτή η προσέγγιση είναι ιδιαίτερα αποδοτική αφού η διαδικασία της αντιγραφής δεδομένων περιορίζεται στο χώρο του χρήστη χωρίς την εμπλοκή του πυρήνα.

Ωστόσο απαιτείται συγχρονισμός της διαδικασίας προσπέλασης των διεργασιών μέσω σημαφόρων για να μην ανακύπτουν συνθήκες ανταγωνισμού.



Διαδιεργασιακή επικοινωνία

Κοινόχρηστη μνήμη

Το Linux προσφέρει δύο διαφορετικά API (Application Programming Interface) για τη χρήση κοινόχρηστης μνήμης, το API του System V και το POSIX API τα οποία δεν είναι συμβατά μεταξύ τους και σε καμία περίπτωση δεν θα πρέπει να συνδυάζονται.

Το POSIX API τελεί ακόμη υπό καθεστώς ανάπτυξης και εξαρτάται από την έκδοση του πυρήνα, κάτι που επηρεάζει και τη φορητότητα.

Το POSIX API υλοποιεί την κοινόχρηστη μνήμη με τη βοήθεια ενός αρχείου που απεικονίζεται στη μνήμη (backing file) και το οποίο περιέχει τα περιεχόμενα του κοινόχρηστου τμήματος μνήμης. Η χρήση αυτού του αρχείου επιτρέπει την επικοινωνία μεταξύ διεργασιών που δεν σχετίζονται μεταξύ τους.

Οι εφαρμογές προσπελαύνουν μόνο το τμήμα της κοινόχρηστης μνήμης και όχι το βοηθητικό αρχείο, ενώ ο συγχρονισμός του αρχείου με την κοινόχρηστη μνήμη γίνεται από το λειτουργικό σύστημα.

Υπάρχουν τρεις τρόποι χρήσης κοινόχρηστης μνήμης και αντίστοιχης επικοινωνίας διεργασιών:

- Μη σχετιζόμενες μεταξύ τους διεργασίες με χρήση βοηθητικού αρχείου.
- Μη σχετιζόμενες μεταξύ τους διεργασίες χωρίς τη χρήση βοηθητικού αρχείου.
- Σχετιζόμενες μεταξύ τους διεργασίες (πατέρας - παιδί) χωρίς τη χρήση βοηθητικού αρχείου.

Διαδιεργασιακή επικοινωνία

<u>Κοινόχρηστη μνήμη</u>

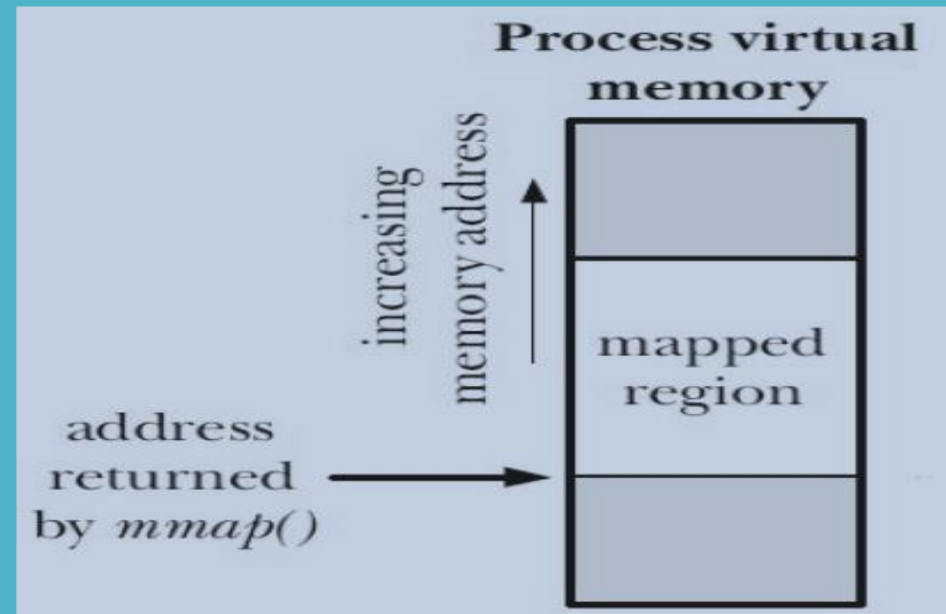
Η βασική συνάρτηση που πραγματοποιεί την παραπάνω διαδικασία είναι η συνάρτηση

**void * mmap (void * daddr, size_t len, int prot,
int flags, int fd, off_t offset)**

με τα ορίσματα που περιέχει να έχουν την ακόλουθη σημασία:

- daddr → ορίζει πού θα τοποθετηθεί η απεικόνιση (εάν αυτό το όρισμα λάβει τιμή NULL αυτή η απόφαση λαμβάνεται από τον πυρήνα).
- len → το μήκος της απεικόνισης σε bytes.
- prot → προστασία μνήμης (read, write, execute)
- flags → MAP_SHARED, MAP_ANONYMOUS
- fd → περιγραφέας βοηθητικού αρχείου
- offset → μετατόπιση μέσα στο βοηθητικό αρχείο.

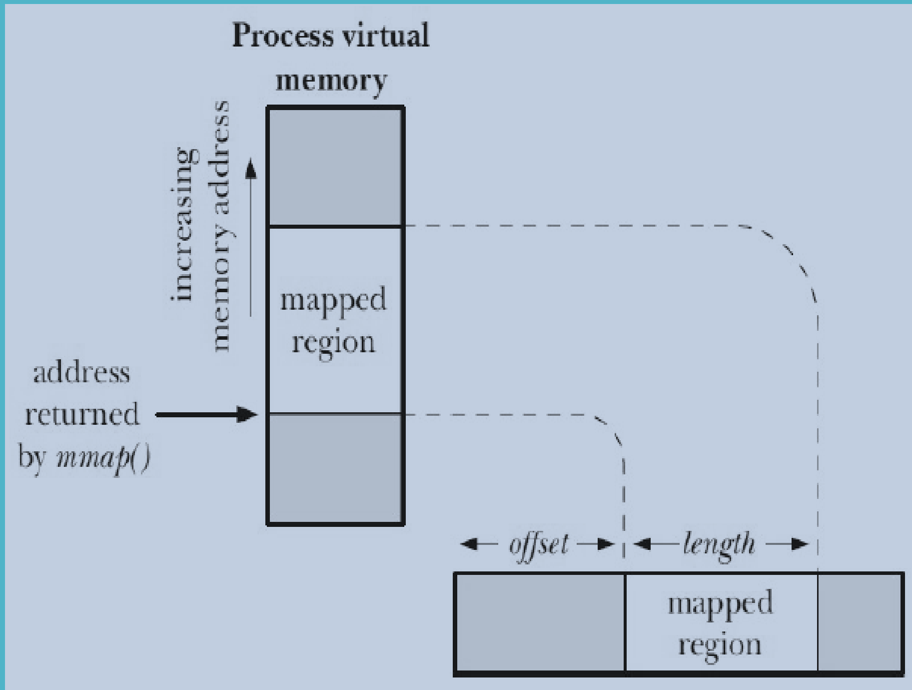
Η επιστρεφόμενη τιμή της συνάρτησης mmap είναι ένας δείκτης που προσδιορίζει το κοινόχρηστο τμήμα μνήμης που ορίζεται με τον παραπάνω τρόπο.

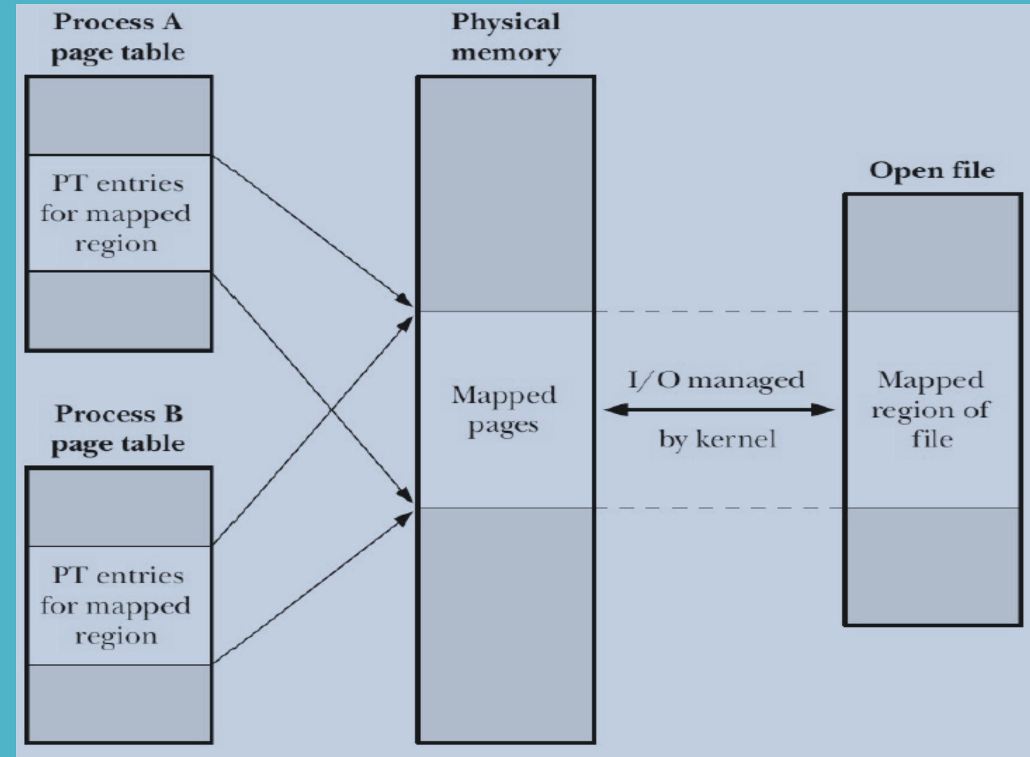


MAP_SHARED → ορατό στις άλλες διεργασίες …

Διαδιεργασιακή επικοινωνία

Κοινόχρηστη μνήμη





Εκτός από τη χρήση κοινόχρηστης μνήμης από σχετιζόμενες διεργασίες, αυτή μπορεί να χρησιμοποιηθεί και από διεργασίες που σχετίζονται μεταξύ τους με σχέση πατέρα – παιδιού.

Ωστόσο, σε αυτή την περίπτωση δεν χρησιμοποιείται βοηθητικό αρχείο και κατά συνέπεια, τα δύο τελευταία ορίσματα στη συνάρτηση nmap έχουν τις τιμές -1 και 0 – επομένως, η συνάρτηση καλείται ως

```
addr = mmap (NULL, len, prot, flags, -1, 0);
```

Διαδιεργασιακή επικοινωνία

Κοινόχρηστη μνήμη

Στο μοντέλο κοινόχρηστης μνήμης του POSIX το σύστημα επιτρέπει την επικοινωνία μη σχετιζόμενων μεταξύ τους διεργασιών χωρίς τη χρήση βοηθητικού αρχείου, με αποτέλεσμα την αύξησης της απόδοσης αφού δεν έχουμε πρόσθετη επιβάρυνση λόγω διαδικασιών εισόδου / εξόδου.

Η δημιουργία / άνοιγμα ενός νέου αντικειμένου ή το άνοιγμα ενός υπάρχοντος αντικειμένου γίνεται με τη βοήθεια της συνάρτησης (απαιτείται η χρήση των αρχείων sys/mman.h, sys/stat.h και fcntl.h)

int shm_open (const char * name, int oflags, mode_t mode)

η οποία επιστρέφει έναν ακέραιο file descriptor (fd) στο νέο αντικείμενο. Στην παραπάνω σύνταξη:

- το όρισμα name (που θα πρέπει να ξεκινά με /) αναφέρεται στο όνομα του αντικειμένου.
- τα oflags είναι παρόμοια με εκείνα της open (O_CREAT, O_EXCL, O_RDONLY, O_RDWR, O_TRUNK)
- το όρισμα mode εκφράζει τα δικαιώματα πρόσβασης.

ΣΗΜΑΝΤΙΚΟ → το βοηθητικό αρχείο (όταν χρησιμοποιείται) δημιουργείται στο σύστημα αρχείων /dev/shm το οποίο ΔΕΝ χρησιμοποιεί το δίσκο αλλά τη μνήμη του υπολογιστή (ramdisk). Αν και αυτό το σύστημα αρχείων είναι τύπου tmpfs (temporary file system) ωστόσο αυτά τα αρχεία ΔΕΝ θα πρέπει να θεωρηθούν της ίδιας φύσης με τους καταλόγους /tmp και /var/tmp που υπάρχουν στο σκληρό δίσκο.

Διαδιεργασιακή επικοινωνία

Κοινόχρηστη μνήμη

Δημιουργία (O_CREAT) και άνοιγμα νέου αντικειμένου shm γνωστού μεγέθους ίσο με size bytes.

```
fd = shm_open("/myshm", O_CREAT | O_EXCL | O_RDWR, 0600);
ftruncate(fd, size);    // Set size of object
addr = mmap(NULL, size,
            PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

Δημιουργία και άνοιγμα υπάρχοντος αντικειμένου shm άγνωστου μεγέθους.

```
fd = shm_open("/myshm", O_RDWR, 0); // No O_CREAT

// Use object size as length for mmap()

struct stat sb;
fstat(fd, &sb);

addr = mmap(NULL, sb.st_size,
            PROT_READ | PROT_WRITE,MAP_SHARED, fd, 0);
```

Κατάργηση ονόματος με την **int shm_unlink (const char * name);**

Διαδιεργασιακή επικοινωνία

Κοινόχρηστη μνήμη

Ο συγχρονισμός των διεργασιών έτσι ώστε να αποτρέπεται η ταυτόχρονη ενημέρωση της κοινόχρηστης μνήμης από τις διεργασίες του συστήματος, πραγματοποιείται με τη βοήθεια σημαφόρων.

Ένας σημαφόρος είναι μία ακέραια μεταβλητή που παρακολουθείται από τον πυρήνα και ο οποίος μπορεί να λάβει μηδενική τιμή και θετικές τιμές, όχι όμως και αρνητική τιμή.

Υπάρχουν δυο είδη σημαφόρων, οι ανώνυμοι που είναι ενσωματωμένοι στην κοινόχρηστη μνήμη και οι επώνυμοι που είναι ανεξάρτητα αντικείμενα.

Εάν υπάρχουν Ν ταυτόσημοι κοινόχρηστοι πόροι, τότε η αρχική τιμή του σημαφόρου είναι Ν ενώ η μεταβολή της πραγματοποιείται με τη βοήθεια των επόμενων συναρτήσεων (του αρχείου semaphore.h)

int sem_post (sem_t * semp); αύξηση της τιμής κατά 1 (unlock)

int sem_wait (sem_t * semp); μείωση της τιμής κατά 1 (lock)

Εκτός από τους παραπάνω σημαφόρους αρίθμησης ή γενικούς σημαφόρους, το σύστημα υποστηρίζει και δυαδικούς σημαφόρους με τιμές 0 και 1.

Διαδιεργασιακή επικοινωνία

Κοινόχρηστη μνήμη

Για τους ανώνυμους σημαφόρους χρησιμοποιούνται οι συναρτήσεις

```
int sem_init (sem_t * semp, int pshared, unsigned int value);
int sem_destroy (sem_t * semp);
```

Η συνάρτηση sem_init αρχικοποιεί τον σημαφόρο semp στην τιμή value. Το όρισμα pshared παίρνει την τιμή 0 εάν η κοινόχρηστη μνήμη χρησιμοποιηθεί από threads και την τιμή 1 εάν η κοινόχρηστη μνήμη χρησιμοποιηθεί από διεργασίες.

Η συνάρτηση sem_destroy διαγράφει έναν ανώνυμο σημαφόρο που είχε δημιουργηθεί με την sem_init.

Για τους επώνυμους σημαφόρους χρησιμοποιούνται οι συναρτήσεις

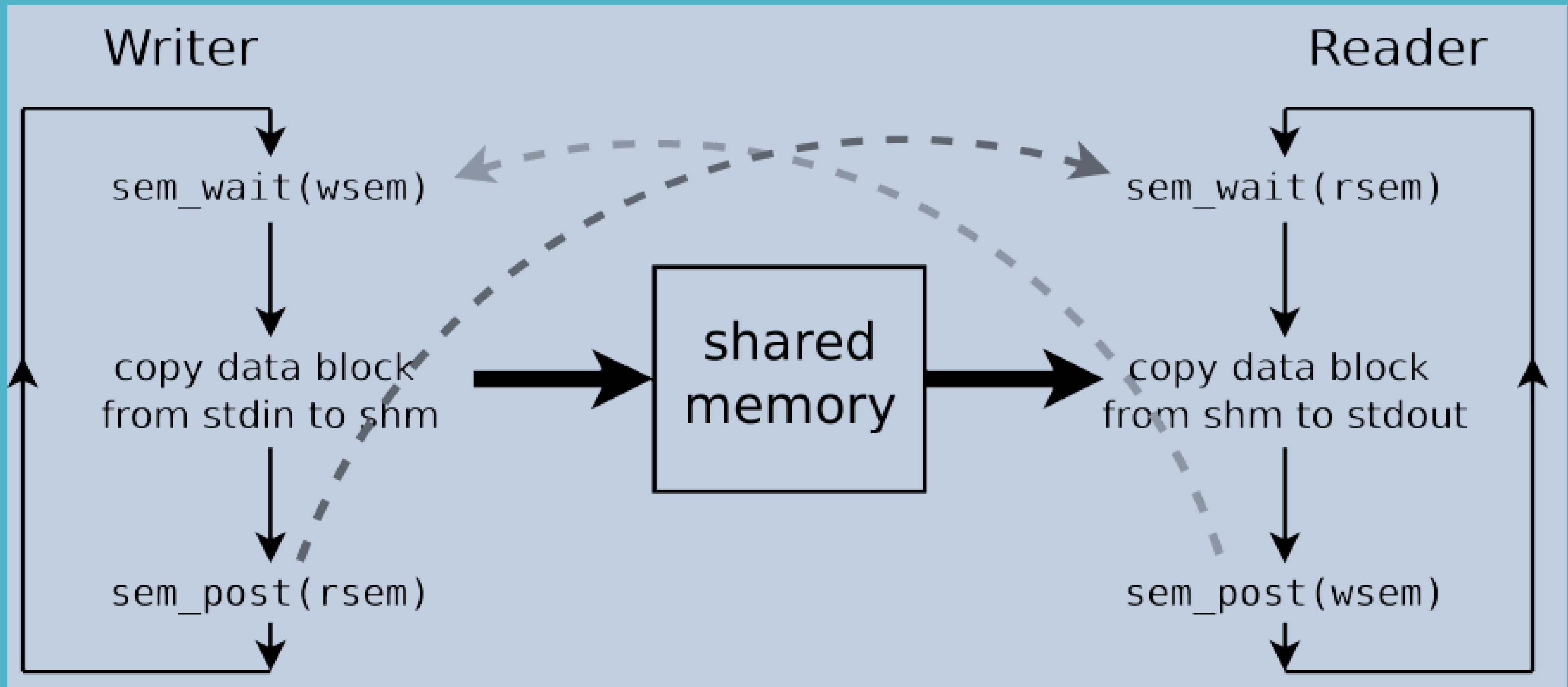
```
int sem_open (const char * name, int oflag,
                   mode_t mode, unsigned int value);

int sem_unlink (const char * name);
```

Διαδιεργασιακή επικοινωνία

<u>Κοινόχρηστη μνήμη</u>

Παράδειγμα χρήσης ανώνυμων σημαφόρων



Διαδιεργασιακή επικοινωνία

Παράδειγμα χρήσης κοινόχρηστης μνήμης και σημαφόρων

Memwriter.c

```
#define ByteSize 512
#define MemContents "This is the way the world ends...\n"

int main() {

  int fd = shm_open("/shMemEx", O_RDWR | O_CREAT, 0644);
  if (fd < 0) { perror ("Can't open shared mem segment..."); exit (-1); }
  ftruncate(fd, ByteSize);

  caddr_t memptr = mmap(NULL, ByteSize, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
  if ((caddr_t)-1==memptr) { perror ("Can't get segment..."); exit (-2); }

  sem_t* semptr = sem_open("sem", O_CREAT, 0644, 0);
  if (semptr==(void*) -1) { perror ("sem_open"); exit (-2); }

  strcpy(memptr, MemContents);

  /* increment the semaphore so that memreader can read */
  if (sem_post(semptr) < 0) { perror ("sem_post"); exit (-3); }

  sleep(12); /* give reader a chance */

  munmap(memptr, ByteSize); /* unmap the storage */
  close(fd);
  sem_close(semptr);
  shm_unlink("/shMemEx");
  return 0; }
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <semaphore.h>
#include <string.h>
```

Αυτή η εφαρμογή γράφει στην κοινόχρηστη μνήμη χρησιμοποιώντας βοηθητικό αρχείο και αναμένει για 12 δευτερόλεπτα για να δώσει τη δυνατότητα στην εφαρμογή ανάγνωσης να διαβάσει τα περιεχόμενα από την κοινόχρηστη μνήμη.

Διαδιεργασιακή επικοινωνία

Παράδειγμα χρήσης κοινόχρηστης μνήμης και σημαφόρων

Memreader.c

```
#define ByteSize 512
#define MemContents "This is the way the world ends...\n"

int main() {

  int fd = shm_open("/shMemEx", O_RDWR, 0644);
  if (fd < 0) { perror ("Can't get file descriptor..."); exit (-1); }

  caddr_t memptr = mmap(NULL, ByteSize, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
  if ((caddr_t)-1==memptr) { perror ("Can't access segment..."); exit (-1); }

  sem_t* semptr = sem_open("sem", O_CREAT, 0644, 0);
  if (semptr == (void*)-1) { perror ("sem open"); exit (-1); }

  if (!sem_wait(semptr)) { /* wait until semaphore != 0 */
    int i;
    for (i = 0; i < strlen(MemContents); i++)
        write(STDOUT_FILENO, memptr + i, 1);
    sem_post(semptr); }

  munmap(memptr, ByteSize);
  close(fd);
  sem_close(semptr);
  unlink(BackingFile);
  return 0; }
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <semaphore.h>
#include <string.h>
```

Αυτή η εφαρμογή εντός δώδεκα δευτερολέπτων από την έναρξη της προηγούμενης, διαβάζει από την κοινόχρηστη μνήμη το περιεχόμενό της και το εκτυπώνει στην οθόνη

Διαδιεργασιακή επικοινωνία

Παράδειγμα χρήσης κοινόχρηστης μνήμης και σημαφόρων

Η εκτέλεση των εφαρμογών η καθεμία από το δικό της ξεχωριστό τερματικό, ακολουθεί στη συνέχεια.

Εάν κατά τη διάρκεια της εκτέλεσης του memwriter ελέγξουμε τα περιεχόμενα του καταλόγου /dev/shm θα διαπιστώσουμε πως σε αυτόν τον κατάλογο υπάρχει το βοηθητικό αρχείο που περιέχει το μήνυμα που ανταλλάσσεται ανάμεσα στις δύο διεργασίες και το οποίο διαγράφεται όταν ολοκληρωθεί η λειτουργία της καθώς και το αρχείο του σεμαφόρου sem.sem.

amarg@amarg-vbox: ~

```
amarg@amarg-vbox:~$ gcc -c memwriter.c
amarg@amarg-vbox:~$ gcc -o memwriter memwriter.o -lrt -lpthread
amarg@amarg-vbox:~$ ./memwriter
amarg@amarg-vbox:~$
amarg@amarg-vbox:~$
```

amarg@amarg-vbox: ~

```
amarg@amarg-vbox:~$ gcc -c memreader.c
amarg@amarg-vbox:~$ gcc -o memwriter memwriter.o -lrt -lpthread
amarg@amarg-vbox:~$ ./memreader
This is the way the world ends...
amarg@amarg-vbox:~$
amarg@amarg-vbox:~$
```

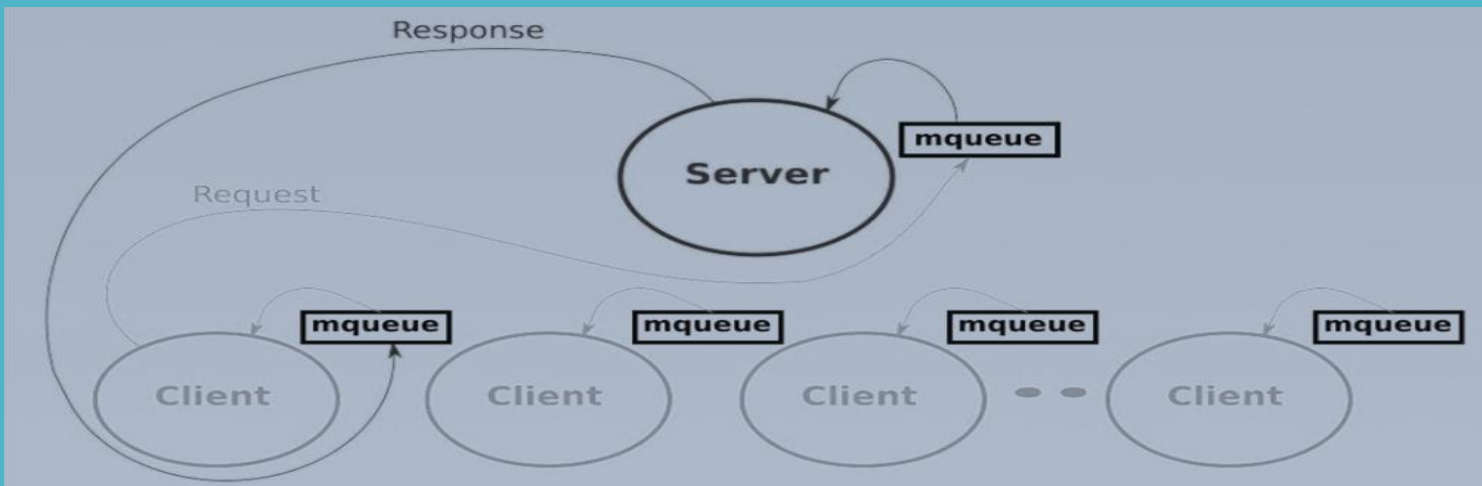
```
amarg@amarg-vbox:/dev/shm$ ls -l
total 8
-rw-r--r-- 1 amarg amarg  32 Окт  27 21:21 sem.sem
-rw-r--r-- 1 amarg amarg 512 Окт  27 21:21 shMemEx
amarg@amarg-vbox:/dev/shm$ cat shMemEx
This is the way the world ends...
amarg@amarg-vbox:/dev/shm$
```

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

Οι ουρές μηνυμάτων (Message Queues, MQs) αποτελούν έναν μηχανισμό επικοινωνίας μεταξύ διεργασιών, ο οποίος χαρακτηρίζεται από τις επόμενες ιδιότητες:

- Επιτρέπει την επικοινωνία μεταξύ διεργασιών χρησιμοποιώντας κατάλληλα διαμορφωμένα μηνύματα με το ανάλογο περιεχόμενο.
- Ο παραλήπτης διαβάζει ένα μήνυμα κάθε φορά χωρίς να επιτρέπεται η ημιτελής ανάγνωση μηνύματος ή η πολλαπλή ανάγνωση του ιδίου μηνύματος.
- Υποστηρίζεται η περίπτωση πολλαπλών αναγνωστών και πολλαπλών συγγραφέων.
- Υποστηρίζεται η χρήση τιμών προτεραιότητας για τα μηνύματα.
- Υποστηρίζονται μηνύματα ειδοποίησης (notification messages).



Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

Διαφορές μεταξύ αγωγών και ουρών μηνυμάτων

- Οι ουρές μηνυμάτων έχουν εσωτερική δομή ενώ στους αγωγούς ο συγγραφέας απλά γράφει bits. Για έναν αναγνώστη, οι διαφορετικές κλήσεις της write από διάφορους συγγραφείς είναι ταυτόσημες και ο προγραμματιστής πρέπει να διασφαλίσει πως γράφεται ή διαβάζεται ο σωστός αριθμός από bits, κάτι που καθιστά δύσκολο τον προγραμματισμό εφαρμογών για μηνύματα διαφορετικού μεγέθους.

- Στις ουρές μηνυμάτων έχουμε τιμές προτεραιότητας και τα μηνύματα είναι ταξινομημένα έτσι ώστε το παλαιότερο μήνυμα με την υψηλότερη τιμή προτεραιότητας να είναι πάντοτε πρώτο.

- Ο προγραμματιστής μπορεί να ορίσει τόσο το μέγιστο πλήθος μηνυμάτων που τοποθετούνται στην ουρά όσο και το μέγεθος κάθε τέτοιου μηνύματος.

- Σε αντίθεση με τους αγωγούς όπου η κατάστασή τους δεν είναι γνωστή, στις ουρές μηνυμάτων η εφαρμογή μπορεί να ανακτήσει τέτοιου είδους πληροφορίες, όπως το πλήθος των μηνυμάτων μιας ουράς, οι μέγιστες τιμές των παραμέτρων που έχουν οριστεί, καθώς και το πλήθος των διεργασιών που έχουν μπλοκάρει κατά την πραγματοποίηση μιας διαδικασίας αποστολής ή παραλαβής.

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

- Στις ουρές μηνυμάτων μπορούν να χρησιμοποιηθούν όλες οι παραδοσιακές συναρτήσεις εισόδου / εξόδου που χρησιμοποιούνται και στα αρχεία (read, write, open, select, κ.τ.λ) αν και διαθέτουν και δικές του συναρτήσεις οι οποίες ξεκινούν με το πρόθεμα mq_ (από το message queue) και είναι δηλωμένες στο αρχείο **mqueue.h**.

- Οι συναρτήσεις διαχείρισης μίας ουράς μηνυμάτων στο POSIX ομαδοποιούνται σε τρεις διαφορετικές κατηγορίες:

  - Συναρτήσεις διαχείρισης της ουράς μηνυμάτων:

    mq_open, mq_close, mq_unlink

  - Συναρτήσεις εισόδου / εξόδου:

    mq_send, mq_receive

  - Βοηθητικές συναρτήσεις:

    mq_setattr, mq_getattr, mq_notify

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

- Η δημιουργία και το άνοιγμα μίας νέας ουράς μηνυμάτων ή το άνοιγμα μιας υπάρχουσας ουράς μηνυμάτων γίνεται με τη συνάρτηση

mqd_t mq_open (const char * name, in oflag,
mode_t mode, struct mq_attr * attr)

ή την πιο απλή εκδοχή της **mqd_t mq_open (const char * name, in oflag)**

Το όρισμα name όπως και στην περίπτωση της κοινόχρηστης μνήμης έχει τη μορφή /somename, το oflag έχει υποχρεωτικά ακριβώς μία από τις τιμές O_RDONLY, O_WRONLY ή O_RDWR και προαιρετικά κάποια από τις τιμές O_CLOEXEC, O_CREAT και O_NONBLOCK.

Το όρισμα mode αναφέρεται στα δικαιώματα πρόσβασης π.χ. 0644, ενώ attr είναι μία δομή τύπου mq_attr – εάν λάβει τιμή NULL χρησιμοποιούνται προεπιλεγμένες τιμές για τις ιδιότητες της ουράς.

```
struct mq_attr {
    long mq_flags;       /* Flags: 0 or O_NONBLOCK */
    long mq_maxmsg;      /* Max. # of messages on queue */
    long mq_msgsize;     /* Max. message size (bytes) */
    long mq_curmsgs;     /* # of messages currently in queue */
};
```

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

Διαπιστώνουμε λοιπόν πως η ουρά μηνυμάτων έχει περιορισμένη χωρητικότητα η τιμή της οποίας ελέγχεται από τις συναρτήσεις διαχείρισης των ιδιοτήτων της ουράς.

Παράδειγμα χρήσης της mq_open

```
// Create new MQ, exclusive,for writing
mqd = mq_open("/mymq",O_CREAT| O_EXCL | O_WRONLY,0600, NULL);
// Open existing queue for reading
mqd = mq_open("/mymq", O_RDONLY);
```

Απομάκρυνση του ονόματος name της ουράς μυνημάτων

int mq_unlink (const char * name)

Το όνομα απομακρύνεται όταν η χρήση της ουράς μηνυμάτων έχει ολοκληρωθεί από όλους τους χρήστες που τη χρησιμοποιούν για επικοινωνία δεδομένων.

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

Αποστολή μηνύματος στην ουρά μηνυμάτων → Πραγματοποιείται με τη συνάρτηση

```
int mq_send (mqd_t mqdes, const char * msg_ptr,
             size_t msg_len, unsigned int msg_prio);
```

όπου mqdes ο περιγραφέας της ουράς μηνυμάτων, msg_ptr ένας δείκτης στα bytes που σχηματίζουν το μήνυμα, msg_len το μέγεθος του μηνύματος και msg_prio η προτεραιότητα του μηνύματος η οποία λαμβάνει μη αρνητική ακέραια τιμή, με την τιμή 0 να περιγράφει τη μικρότερη δυνατή προτεραιότητα.

Το μέγεθος του μηνύματος msg_len πρέπει να είναι μικρότερο ή ίσο της τιμής της παραμέτρου mq_msgsize της ουράς. Η αποστολή μηνύματος μηδενικού μεγέθους, είναι επιτρεπτή.

Η συνάρτηση mq_send μπλοκάρει όταν η ουρά μηνυμάτων είναι γεμάτη και δεν μπορεί να φιλοξενήσει άλλα μηνύματα. Εάν θέλουμε να λειτουργήσει η συνάρτηση σε non-blocking mode θα πρέπει στη συνάρτηση mq_open να χρησιμοποιήσουμε το flag O_NONBLOCK.

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

Παραλαβή μηνύματος από την ουρά μηνυμάτων → Πραγματοποιείται με τη συνάρτηση

```
ssize_t mq_receive (mqd_t mqdes, char * msg_ptr,
                    size_t msg_len, unsigned int * msg_prio);
```

η οποία απομακρύνει από την ουρά μηνυμάτων το παλαιότερο μήνυμα με την υψηλότερη προτεραιότητα και το τοποθετεί στην περιοχή μνήμης που προσδιορίζεται από τον msg_ptr, το μέγεθος της οποίας θα πρέπει να είναι μεγαλύτερο ή ίσο της τιμής της παραμέτρου mq_msgsize της ουράς.

Εάν η τιμή της προτεραιότητας δεν είναι NULL, αυτή επιστρέφεται στο όρισμα msg_prio. Η συνάρτηση mq_receive μπλοκάρει όταν επιχειρεί να παραλάβει μήνυμα από μία κενή ουρά. Για λειτουργία σε non blocking mode χρησιμοποιείται όπως και πριν το flag O_NONBLOCK.

Σε περίπτωση επιτυχούς εκτέλεσης, η συνάρτηση mq_receive επιστρέφει το πλήθος των bytes που υπάρχουν στο μήνυμα που παρέλαβε, ενώ αντίθετα η συνάρτηση mq_send επιστρέφει μηδενική τιμή. Σε περίπτωση αποτυχίας, αμφότερες οι συναρτήσεις επιστρέφουν την τιμή -1 με την μεταβλητή errno να τίθεται στην κατάλληλη τιμή σφάλματος.

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

Παραδείγματα χρήσης των συναρτήσεων mq_send και mq_receive

```
mqd_t mqd;
mqd = mq_open("/mymq",O_CREAT | O_WRONLY, 0600, NULL);
char *msg = "hello world";
mq_send(mqd, msg, strlen(msg), 0);
```

```
const int BUF_SIZE = 1000;
char buf[BUF_SIZE];
unsigned int prio;
...
mqd = mq_open("/mymq", O_RDONLY);
nbytes = mq_receive(mqd, buf,BUF_LEN, &prio);
```

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

Ειδοποίηση για την άφιξη μηνύματος προς παραλαβή

int mq_notify (mqd_t mqdes, const struct sigevent * sevp);

Μέσω της συνάρτησης mq_notify η διεργασία γνωστοποιεί πως επιθυμεί να ειδοποιηθεί όταν ένα νέο μήνυμα φτάσει σε μία κενή ουρά μηνυμάτων. Σε αυτή τη συνάρτηση χρησιμοποιείται η δομή

```
union sigval {          /* Data passed with notification */
    int     sival_int;         /* Integer value */
    void   *sival_ptr;         /* Pointer value */
};

struct sigevent {
    int          sigev_notify; /* Notification method */
    int          sigev_signo;  /* Notification signal */
    union sigval sigev_value;  /* Data passed with notification */

    void       (*sigev_notify_function) (union sigval); /* Function used for thread notification (SIGEV_THREAD) */
    void        *sigev_notify_attributes; /* Attributes for notification thread (SIGEV THREAD) */
    pid_t        sigev_notify_thread_id;/* ID of thread to signal (SIGEV_THREAD_ID) */
};
```

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

Ορισμός και λήψη τιμών ιδιοτήτων

int mq_getattr (mqd_t mqdes, struct mq_attr * attr);

int mq_setattr (mqd_t mqdes,
const struct mq_attr * newattr, struct mq_attr * newattr);

Οι παραπάνω συναρτήσεις επιστρέφουν 0 στην περίπτωση επιτυχούς εκτέλεσης και -1 σε περίπτωση αποτυχίας, αρχικοποιώντας κατάλληλα και τη μεταβλητή errno.

Η δομή mq_attr όπως σχολιάσαμε και προηγουμένως, έχει τη μορφή

```
struct mq_attr {
    long mq_flags;          /* Flags: 0 or O_NONBLOCK */
    long mq_maxmsg;         /* Max. # of messages on queue */
    long mq_msgsize;        /* Max. message size (bytes) */
    long mq_curmsgs;        /* # of messages currently in queue */
};
```

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

```
#include <mqueue.h>
#include <limits.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <signal.h>
#include <string.h>

#define MSG_SIZE 16384
```

```
void handler (int sig_num) { printf ("Received sig %d.\n", sig_num); }

int main (int argc, char * argv []) {
  struct mq_attr attr, old_attr;
  struct sigevent sigevent;
  mqd_t mqdes, mqdes2;
  char * message = "Hello world";
  char buf [MSG_SIZE];
  unsigned int prio=0, i;
  mqdes = mq_open ("/mqueue1", O_RDWR | O_CREAT, 0664, NULL);
  mq_getattr (mqdes, &attr);
  printf ("Max number of messages on the queue ==> %ld messages.\n", attr.mq_maxmsg);
  printf ("Size of messages on the queue ==> %ld bytes.\n", attr.mq_msgsize);
  printf ("%ld messages are currently on the queue.\n", attr.mq_curmsgs);
  if (attr.mq_curmsgs != 0) {                                              ⎫
    attr.mq_flags = O_NONBLOCK;                                            ⎪
    mq_setattr (mqdes, &attr, &old_attr);                                  ⎪
    while (mq_receive (mqdes, &buf[0], MSG_SIZE, &prio) != -1)             ⎬ A
           printf ("Received a message with priority %d.\n", prio);      ⎪
    if (errno != EAGAIN) { perror ("mq_receive()"); exit (EXIT_FAILURE); } ⎪
    mq_setattr (mqdes, &old_attr, 0); }                                    ⎭
  signal (SIGUSR1, handler);
  sigevent.sigev_signo = SIGUSR1;
  if (mq_notify (mqdes, &sigevent) == -1) {
    if (errno == EBUSY)
        printf ("Another process has registered for notification.\n");
        exit (EXIT_FAILURE); }
  for (i= 0; i < attr.mq_maxmsg; i++) {                                    ⎫
      printf ("Writing a message with priority %d.\n", prio);              ⎬ B
      if (mq_send (mqdes, message, strlen(message)+1, prio) == -1) perror ("mq_send()");
      prio += 5;}                                                          ⎭
  mq_close (mqdes);
  return (0); }
```

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

Αρχικά η ουρά μηνυμάτων είναι κενή και το τμήμα κώδικα **A** δεν εκτελείται, αφού δεν υπάρχουν μηνύματα προς παραλαβή.

Εκτελείται λοιπόν το τμήμα κώδικα **B**, μέσω του οποίου αποστέλλονται στην ουρά μηνυμάτων το μέγιστο πλήθος μηνυμάτων που αυτή μπορεί να φιλοξενήσει που είναι ίσο με 10 (η προεπιλεγμένη τιμή του πεδίου mq_maxmsg της δομής mq_attr). Η έξοδος της εφαρμογής σε αυτή την πρώτη εκτέλεση του κώδικα έχει τη μορφή

```
Max number of messages on the queue ==> 10 messages.
Size of messages on the queue ==> 8192 bytes.
0 messages are currently on the queue.
Writing a message with priority 0.
Received sig 10.
Writing a message with priority 5.
Writing a message with priority 10.
Writing a message with priority 15.
Writing a message with priority 20.
Writing a message with priority 25.
Writing a message with priority 30.
Writing a message with priority 35.
Writing a message with priority 40.
Writing a message with priority 45.
```

Μετά την τοποθέτηση στην κενή ουρά του πρώτου μηνύματος, στάλθηκε στη διεργασία το σήμα SIGUSR1 με τιμή 10 επειδή η διεργασία ζήτησε να ειδοποιηθεί όταν συμβεί κάτι τέτοιο.

```
signal (SIGUSR1, handler);
sigevent.sigev_signo = SIGUSR1;
if (mq_notify (mqdes, &sigevent)
  if (errno == EBUSY)
```

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

Αυτή η διαδικασία δημιουργεί στον κατάλογο /dev/mqueue το αρχείο mqueue1 τα περιεχόμενα του οποίου ακολουθούν στη συνέχεια.

```
amarg@amarg-vbox:/dev/mqueue$ ls -l
total 0
-rw-rw-r-- 1 amarg amarg 80 Noε   1 14:33 mqueue1
amarg@amarg-vbox:/dev/mqueue$ cat mqueue1
QSIZE:120         NOTIFY:0     SIGNO:0     NOTIFY_PID:0

amarg@amarg-vbox:/dev/mqueue$
```

Το πεδίο QSIZE : 120 σε αυτό το αρχείο υποδηλώνει πως στην ουρά mqueue1 υπάρχουν 120 bytes προς παραλαβή [10 messages x strlen (“Hello world”)]= 10 x 12 = 120 τα οποία έχουν παραληφθεί από τον πυρήνα και αναμένουν να διαβαστούν.

Προκειμένου να διαβάσουμε τα 10 μηνύματα που στάλθηκαν στην ουρά μηνυμάτων, εκτελούμε την εφαρμογή για δεύτερη φορά. Σε αυτή τη δεύτερη εκτέλεση, επειδή υπάρχουν μηνύματα για παραλαβή, θα εκτελεστεί αρχικά το τμήμα Α του κώδικα προκειμένου η διεργασία να διαβάσει τα 10 αυτά μηνύματα και στη συνέχεια το τμήμα Β προκειμένου η διεργασία να αποστείλει στην ουρά 10 νέα μηνύματα.

Κατά συνέπεια, η έξοδος της εφαρμογής σε αυτή τη δεύτερη εκτέλεση θα έχει τη μορφή

Διαδιεργασιακή επικοινωνία

Ουρές μηνυμάτων στο POSIX

Παρατηρήστε πως τα μηνύματα έχουν διαταχθεί ανάλογα με την τιμή της προτεραιότητάς τους και κατά συνέπεια τα μηνύματα παρελήφθησαν έτσι ώστε αυτά που χαρακτηρίζονται από μεγαλύτερη τιμή προτεραιότητας να παραληφθούν πρώτα.

Μετά την παραλαβή των μηνυμάτων που υπάρχουν στην ουρά, η διεργασία αποστέλλει στην ουρά μηνυμάτων 10 νέα μηνύματα και κατά συνέπεια το αρχείο /dev/mqueue/mqueue1 είναι το ίδιο με πριν.

Σε αυτό το παράδειγμα για λόγους απλότητας το όρισμα attr τέθηκε στην τιμή NULL έτσι ώστε να χρησιμοποιηθούν οι προεπιλεγμένες τιμές (μέγιστο πλήθος μηνυμάτων ίσο με 10 με μέγεθος μηνύματος ίσο με 8192 bytes αλλά σε κάθε περίπτωση μπορούμε μέσω του ορίσματος attr να ορίσουμε τις τιμές των παραμέτρων της ουράς).

```
amarg@amarg-vbox:~$ ./mqExample1
Max number of messages on the queue ==> 10 messages.
Size of messages on the queue ==> 8192 bytes.
10 messages are currently on the queue.
Received a message with priority 45.
Received a message with priority 40.
Received a message with priority 35.
Received a message with priority 30.
Received a message with priority 25.
Received a message with priority 20.
Received a message with priority 15.
Received a message with priority 10.
Received a message with priority 5.
Received a message with priority 0.
Writing a message with priority 0.
Received sig 10.
Writing a message with priority 5.
Writing a message with priority 10.
Writing a message with priority 15.
Writing a message with priority 20.
Writing a message with priority 25.
Writing a message with priority 30.
Writing a message with priority 35.
Writing a message with priority 40.
Writing a message with priority 45.
amarg@amarg-vbox:~$
```