ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΑΚΑΔΗΜΑΪΚΟ ΕΤΟΣ 2016-2017

ΧΕΙΜΕΡΙΝΟ ΕΞΑΜΗΝΟ

ΜΑΘΗΜΑ: ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

ΔΙΔΑΣΚΩΝ: ΚΟΛΟΜΒΑΤΣΟΣ ΚΩΝ/ΝΟΣ

ΕΡΓΑΣΤΗΡΙΟ 5 - ΣΗΜΕΙΩΣΕΙΣ

Σημαφόροι

Όταν εργαζόμαστε με προγράμματα που διαχειρίζονται πολλαπλές διεργασίες ή νήματα θα πρέπει να αναγνωρίζουμε τις κρίσιμες περιοχές του κώδικα ώστε να εξασφαλίζουμε ότι μόνο μια διεργασία – νήμα εκτελεί το συγκεκριμένο τμήμα κώδικα. Για την υλοποίηση του συγχρονισμού και την αποκλειστικπή πρόσβαση στις κρίσιμες περιοχές, μπορύμε να υιοθετήσουμε τη λύση των σημαφόρων.

Ένας σημαφόρος είναι μια ειδική μεταβλητή για την οποία μόνο δύο λειτουργίες επιτρέπονται: wait (αναμονή) και signal (σήμα). Χρησιμοποιούμε τους σημαφόρους:

- P για wait
- V για signal

Τα ονόματα βγαίνουν από τις λέξεις passeren: to pass / proberen: to test, to try και vrijgeven: to give or release / verhogen: increase. Επίσης, χρησιμοποιούνται ευρέως οι όροι Up και Down.

Ο πιο απλός σημαφόρος είναι μια μεταβλητή που παίρνει τις τιμές 0 ή 1 (δυαδικός σημαφόρος). Ας υποθέσουμε ότι έχουμε ένα σημαφόρο sv. Ο ακόλουθος πίνακας παρουσιάζει τη σημασία των ενεργειών Up και Down:

`P(sv)`	If `sv` is greater than zero, decrement `sv`. If `sv` is zero, suspend execution of this process.
`V(sv)`	If some other process has been suspended waiting for `sv`, make it resume execution. If no process is suspended waiting for `sv`, increment `sv`.

Αν λοιπόν ο σημαφόρος είναι true (1) όταν η κρίσιμη περιοχή είναι προς εκτέλεση, ο σημαφόρος μειώνει κατά 1 με την P(sv) οπότε γίνεται false. Όταν μια διεργασία έχει εκτελέσει το P(sv) τότε 'αποκτά' τον σημαφόρο και μπορεί να εκτελέσει την κρίσιμη περιοχή, συνεπώς αποκλείονται όλες οι άλλες διεργασίες. Οι υπόλοιπες αποκλείονται μέχρι να απελευθερωθεί ο σημαφόρος από τη διεργασία που τον κατέλαβε. Το επόμενο παράδειγμα μας δείχνει ένα αλγόριθμο εκτέλεσης της διαδιακασίας.

```
semaphore sv = 1;

loop forever {
    P(sv);
    critical code section;
    V(sv);
    noncritical code section;
}
```

Στο Linux υπάρχει διαθέσιμο ένα σύνολο συναρτήσεων για χειρισμό σημαφόρων. Αυτές είναι οι εξής:

```
#include <sys/sem.h>

int semctl(int sem_id, int sem_num, int command, ...);
int semget(key_t key, int num_sems, int sem_flags);
int semop(int sem_id, struct sembuf *sem_ops, size_t num_sem_ops);
```

Συνάρτηση semget

Η συνάρτηση semget δημιουργεί ένα νέο σημαφόρο ή αποκτά το κλειδί ενός υπάρχοντος σημαφόρου. Οι πάράμετροί της έχουν ως εξής:

- key: επιτρέπει διαφορετικές διεργασίες να προσπελάσουν το σημαφόρο. Η αναφορά IPC_PRIVATE δημιουργεί το σημαφόρο με πρόσβαση μόνο από τη διεργασία που τον δημιούργησε.
- num_sems: περιγράφει τον αριθμό των σημαφόρων (συνήθως 1).
- sem_flags: ένα σύνολο από flags όπου δικαιώματα και πράξεις πρόσβασης (οι δυνατές πράξεις είναι IPC_CREAT ή IPC_EXCL). Συνήθως θέτουμε αυτό το flag στην τιμή IPC_CREAT | 0660. Μπορούμε να χρησιμοποιήσουμε τα IPC_CREAT και IPC_EXCL για να δημιουργήσουμε ένα νέο μοναδικό σημαφόρο (επιστροφή σφάλαματος αν ο σημαφόρος υπάρχει).

Συνάρτηση semop

Η συνάρτηση semop χρησιμοποιείται για την αλλαγή της τιμής ενός σημαφόρου. Παράμετροι:

- semid: το id που επέστρεψε η semget.
- sem_ops: η διεύθυνση ενός πίνακα που περιέχει τις πράξεις που θα εφαρμοστούν στους σημαφόρους
- num_sem_ops: το μέγεθος του προηγούμενου πίνακα

Ο πίνακας που χρησιμοποιείται στο δεύτερο όρισμα αποτελείται από τις παρακάτω δομές. Κάθε μια από αυτές τις δομές καθορίζει μια πράξη πάνω σε ένα σημαφόρο:

```
struct sembuf {
        short sem_num;
        short sem_op;
        short sem_flg;
};
```

- sem_num: καθορίζει σε ποιον σημαφόρο από αυτούς που ορίσαμε στην semget θα εφαρμοστεί η πράξη (άρα μπορεί να πάρει τιμές από 0 έως nsems-1)
- sem_op: το θέτουμε <0 (συνήθως -1) για δέσμευση ενός σημαφόρου και >0 (συνήθως 1) για αποδέσμευση
- sem_flg: συνήθως το θέτουμε ίσο με 0

Συνάρτηση semctl

Η κλήση συστήματος sem_ctl επιτρέπει τον απ' ευθείας έλεγχο των πληροφοριών των σημαφόρων. Παράμετροι:

- semid: το id που επέστρεψε η semget
- semnum: ο σημαφόρος του συνόλου στον οποίο θα εφαρμοστεί η πράξη (0 έως nsems-1)
- cmd: η πράξη που θέλουμε να εφαρμόσουμε στον σημαφόρο (πιθανές πράξεις είναι οι SETVAL, GETVAL, SETALL, GETALL, IPC_RMID).
- arg: χρησιμοποιείται σε συνδυασμό με την πράξη. Ουσιαστικά περιέχει τα ορίσματα της πράξης. Οι δύο πρώτες πράξεις (που είναι και οι πιο συνηθισμένες μαζί με την τελευταία) χρησιμοποιούν το πεδίο val της union senum, η οποία ορίζεται παρακάτω (ο ορισμός της βρίσκεται στο sem.h. Αν δεν χρησιμοποιούμε Linux μπορούμε να την ορίσουμε εμείς οι ίδιοι στο κώδικά μας όπως φαίνεται και παρακάτω):

```
union senum {
        int val;
        struct semid ds *buff;
        unsigned short *array;
```

};

Μια συνηθισμένη πράξη της semctl είναι η αρχικοποίηση των σημαφόρων.
Για να αρχικοποιήσουμε το 3ο σημαφόρο ενός συνόλου σημαφόρων, που δημιουργήσαμε με την semget, στην τιμή 1 θα χρησιμοποιούσαμε τις εντολές:

union senum arg;
arg.val=1;
semctl(semid, 2, SETVAL, arg)

Βέβαια δεν είναι υποχρεωτικό να αρχικοποιηθεί κάποιος σημαφόρος. Τότε όμως τι τιμή θα έχει; (αφήνεται ως άσκηση)

Για να διαγράψουμε ένα σημαφόρο θα χρησιμοποιούσαμε την εντολή:

semctl(semid, 0, IPC_RMID,0)

Προσοχή: Πάντα να ελέγχονται οι τιμές που επιστρέφουν οι κλήσεις συστήματος και τα τυχόν σφάλματα καλό είναι να εμφανίζονται στην οθόνη με την συνάρτηση perror(char* errorString).

Code 1

Semun.h

```
union semun {
        int val;
        struct semid_ds *buf;
        unsigned short *array;
}
```

Code

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/sem.h>
#include "semun.h"
static int set_semvalue(void);
static void del_semvalue(void);
static int semaphore_p(void);
static int semaphore_v(void);
static int sem_id;
int main(int argc, char *argv[])
{
        int i;
        int pause_time;
        char op_char = 'O';
        srand((unsigned int)getpid());
        sem_id = semget((key_t)1234, 1, 0666 | IPC_CREAT);
        if (argc > 1) {
                if (!set_semvalue()) {
                        fprintf(stderr, "Failed to initialize semaphore\n");
                        exit(EXIT_FAILURE);
                }
                op_char = 'X';
                sleep(2);
```

```
        }

        for(i = 0; i < 10; i++) {
                if (!semaphore_p()) exit(EXIT_FAILURE);
                printf("%c", op_char);fflush(stdout);
                pause_time = rand() % 3;
                sleep(pause_time);
                printf("%c", op_char);fflush(stdout);
                if (!semaphore_v()) exit(EXIT_FAILURE);
                        pause_time = rand() % 2;
                        sleep(pause_time);
                }
                printf("\n%d - finished\n", getpid());
                if (argc > 1) {
                        sleep(10);
                        del_semvalue();
                }
                exit(EXIT_SUCCESS);
        }

static int set_semvalue(void)
{
        union semun sem_union;
        sem_union.val = 1;
        if (semctl(sem_id, 0, SETVAL, sem_union) == -1) return(0);
        return(1);
}

static void del_semvalue(void)
{
        union semun sem_union;
        if (semctl(sem_id, 0, IPC_RMID, sem_union) == -1)
        fprintf(stderr, "Failed to delete semaphore\n");
}

static int semaphore_p(void)
{
        struct sembuf sem_b;
        sem_b.sem_num = 0;
        sem_b.sem_op = -1; /* P() */
        sem_b.sem_flg = SEM_UNDO;
        if (semop(sem_id, &sem_b, 1) == -1) {
                fprintf(stderr, "semaphore_p failed\n");
                return(0);
        }
        return(1);
}

static int semaphore_v(void)
{
        struct sembuf sem_b;
        sem_b.sem_num = 0;
        sem_b.sem_op = 1; /* V() */
        sem_b.sem_flg = SEM_UNDO;
```

```
        if (semop(sem_id, &sem_b, 1) == -1) {
                fprintf(stderr, "semaphore_v failed\n");
                return(0);
        }
        return(1);
}
```

Code 2

Common.h

```
#ifndef _COMMON_H_
#define _COMMON_H_

#define SEM_KEY_FILE ("sem.key")

#endif /* _COMMON_H_ */
```

Server.c

```
/**
 * Run these files:
 *
 * $ gcc -o sem_server sem_server.c
 * $ gcc -o sem_client sem_client.c
 * $ ./sem_server
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <fcntl.h>
#include <sys/sem.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <errno.h>

#include "common.h"

#define CLIENT_PATH_BUFSIZE 255

int main(int argc, char *argv[]) {
   key_t sem_key;
   int sem_id;
   int sem_fd;
   char client_exe[CLIENT_PATH_BUFSIZE];
   int dir_len;
   int i;
   struct sembuf sop;
   int pid;
   int status;

   sem_key = ftok("./sem_server.c", 42);
   // The ftok() function uses the identity of the file named by the given
  //    pathname (which must refer to an existing, accessible file) and the
```

```
  // least significant 8 bits of proj_id (which must be nonzero) to
  // generate a key_t type System V IPC key, suitable for use with
  // msgget, semget, or shmget.

// Write the key to a file for children to pick it up
sem_fd = open(SEM_KEY_FILE, O_WRONLY | O_TRUNC | O_EXCL | O_CREAT, 0644);
if (sem_fd < 0) {
   perror("Could not open sem.key");
   exit(1);
}

// Actual write of the key
if (write(sem_fd, &sem_key, sizeof(key_t)) < 0) {
   perror("Could not write key to file");
   exit(2);
}

// Done with the key
close(sem_fd);

// Create the semaphore
sem_id = semget(sem_key, 1, IPC_CREAT | IPC_EXCL | 0600);
if (sem_id < 0) {
   perror("Could not create sem");
   unlink(SEM_KEY_FILE);
   exit(3);
}

if (semctl(sem_id, 0, SETVAL, 0) < 0) {
   perror("Could not set value of semaphore");
   exit(4);
}

// Now create some clients
// First create the path to the client exec
getcwd(client_exe, CLIENT_PATH_BUFSIZE);
dir_len = strlen(client_exe);
strcpy(client_exe + dir_len, "/sem_client");
printf("%s\n", client_exe);

for (i = 0; i < 5; ++i) {
   if ((pid = fork()) < 0) {
      perror("Could not fork, please create clients manually");
   }
   else if (pid == 0) {
      // We're in the child process, start a client
      execl(client_exe, "sem_client", (char*)0);
      _exit(127);
   }
}

printf("Done creating clients, sleeping for a while\n");
sleep(5);
printf("Increasing sem count\n");
```

```
    sop.sem_num = 0;
    sop.sem_op = 1;
    sop.sem_flg = 0;
    if (semop(sem_id, &sop, 1)) {
        perror("Could not increment semaphore");
        exit(5);
    }

    // Wait for all children to finish
    for (;;) {
        // Remove the zombie process, and get the pid and return code
        pid = wait(&status);
        if (pid < 0) {
            if (errno == ECHILD) {
                printf("All children have exited\n");
                break;
            }
            else {
                perror("Could not wait");
            }
        }
        else {
            printf("Child %d exited with status %d\n", pid, status);
        }
    }

    // Delete semaphore and file
    if (unlink(SEM_KEY_FILE) < 0) {
        perror("Could not unlink key file");
    }

    if (semctl(sem_id, 0, IPC_RMID) < 0) {
        perror("Could not delete semaphore");
    }

    exit(0);
}
```

Client.c

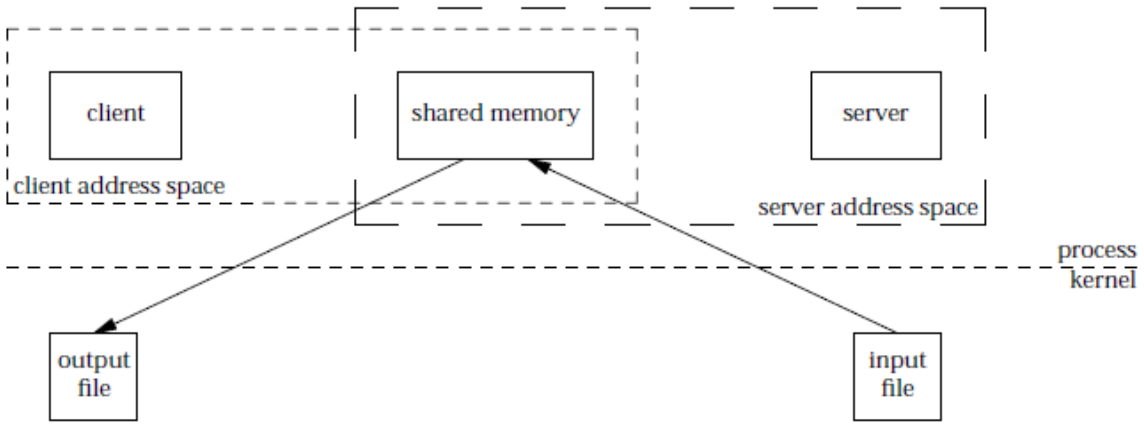
```
/**
 * Compile this file as "sem_client" for the pair of programs to work properly
 * (view the info on sem_server.c on how to run this)
 *
 * gcc -o sem_client sem_client.c -Wall -Werror
 *
 * This is needed because the server does an `exec` of this program
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/sem.h>
#include <fcntl.h>
#include <unistd.h>

#include "common.h"
```

```
int main(int argc, char *argv[]) {
   int sem_fd;
   key_t sem_key;
   int sem_id;
   int i;
   struct sembuf sop;

   // Recover the sem_key from file
   sem_fd = open(SEM_KEY_FILE, O_RDONLY);
   if (sem_fd < 0) {
      perror("Could not open sem key for reading");
      exit(1);
   }

   // Technically speaking, the read could read less than sizeof(key_t)
   // Which would be wrong.
   // But in our case, it is not likely to happen...
   if (read(sem_fd, &sem_key, sizeof(key_t)) != sizeof(key_t)) {
      perror("Error reading the semaphore key");
      exit(2);
   }

   // Done getting the semaphore key
   close(sem_fd);

   // Now obtain the (hopefully) existing sem
   sem_id = semget(sem_key, 0, 0);
   if (sem_id < 0) {
      perror("Could not obtain semaphore");
      exit(3);
   }

   for (i = 0; i < 5; ++i) {
      sop.sem_num = 0;
      sop.sem_op = -1;
      sop.sem_flg = SEM_UNDO;
      printf("Client #%d waiting\n", getpid());
      semop(sem_id, &sop, 1);
      printf("Client #%d acquired. Sleeping\n", getpid());
      sleep(1);
      printf("Client #%d releasing\n", getpid());
      sop.sem_op = 1;
      semop(sem_id, &sop, 1);
   }

   exit(0);
}
```

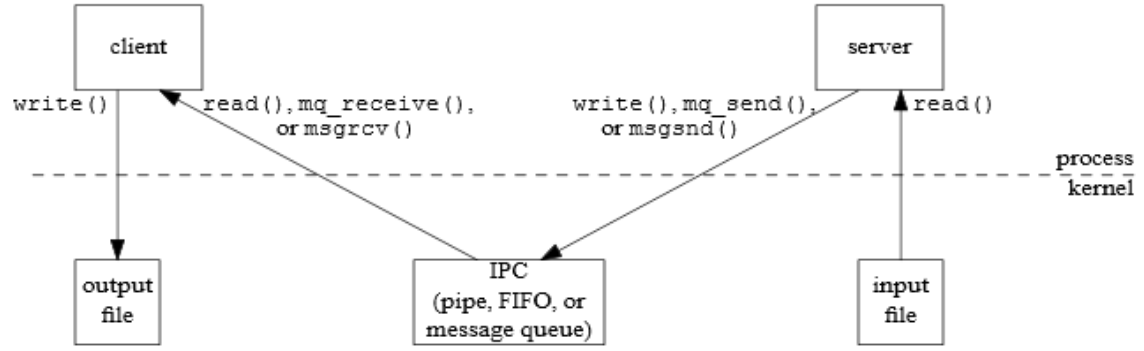
Διαμοιραζόμενη Μνήμη

Η διαμοιραζόμενη μνήμη επιτρέπει δύο διεργασίες να αποκτήσουν πρόσβαση στο ίδιο τμήμα λογικής μνήμης. Η χρήση της διαμοιραζόμενης μνήμης κρίνεται αποδοτική όταν επιθυμούμε να διαμοιράσουμε δεδομένα ανάμεσα σε ένα σύνολο διεργασιών. Μια διαμοιραζόμενη μνήμη

είναι μια ακολουθία διευθύνσεων που παρουσιάζονται στο χώρο διευθύνσεων μιας διεργασίας. Άλλες διεργασίες μπορούν να 'επισυνάψουν' (attach) τις ίδιες διευθύνσεις στο δικό τους χώρο διευθύνσεων. Με αυτό τον τρόπο αν μια διεργασία γράψει στο χώρο αυτό, τότε οι υπόλοιπες άμεσα έχουν πρόσβαση στα δεδομένα. Η ίδια η διαμοιραζόμενη μνήμη δεν παρέχει δυνατότητες συγχρονισμού των διεργασιών. Προφανώς, χρειαζόμαστε κάποιο μηχανισμό που να επιτρέπει το συγχρονισμό των διεργασιών. Η ακόλουθη εικόνα μας παρουσιάζει το μηχανισμό επικοινωνίας δύο διεργασιών με χρήση διαμοιραζόμενης μνήμης.



Γενικά, η επικοινωνία με τη βοήθεια διαμοιραζόμενης μνήμης είναι πιο γρήγορή σε αντίθεση με τις σωληνώσεις και ουρές μηνυμάτων. Η ακόλουθη εικόνα παρουσιάζει το μηχανισμό επικοινωνίας ανάμεσα σε διεργασίες χωρίς τη χρήση διαμοιραζόμενης μνήμης.



Οι συναρτήσεις που μπορούν να χρησιμοποιηθούν για τη διαχείριση διαμοιραζόμενης μνήμης είναι οι ακόλουθες:

```
#include <sys/shm.h>

void *shmat(int shm_id, const void *shm_addr, int shmflg);
int shmctl(int shm_id, int cmd, struct shmid_ds *buf);
int shmdt(const void *shm_addr);
int shmget(key_t key, size_t size, int shmflg);
```

Όπως και με τους σημαφόρους οι βιβλιοθήκες sys/types.h και sys/ipc.h περιλαμβάνονται αυτόματα με τη χρήση της shm.h.

Συνάρτηση shmget

Η συνάρτηση shmget χρησιμοποιείται για τη δημιουργία ενός τμήματος διαμοιραζόμενης μνήμης. Η συνάρτηση επιστρέφει ένα ID της μνήμης με το οποίο αναφερόμαστε στο υπόλοιπο τμήμα του προγράμματος. Παράμετροι:

- key: είναι το όνομα / κλειδί της διαμοιραζόμενης μνήμης. Υπάρχει η ειδική τιμή του κλειδιού IPC_PRIVATE το οποίο δημιουργεί ένα τμήμα μνήμης το οποίο είναι ιδιωτικό και ανήκει στη διεργασία που το δημιούργησε.
- size: Καθορίζει την ποσότητα της μνήμης σε bytes.
- shmflg: περιέχει εννέα (9) flags αδειών όπως ακριβώς λειτουργούν οι άδειες κατά τη διαχείριση των αρχείων. Ένα ειδικό bit το IPC_CREAT πρέπει να ORποιηθεί με τις άδειες της νέας μνήμης.

Όταν η διαμοιραζόμενη μνήμη δημιουργηθεί ομαλά, η συνάρτηση επιστρέφει ένα θετικό ακέραιο ενώ σε διαφορετική περίπτωση επιστρέφει το -1.

Συνάρτηση shmat

Η συνάρτηση shmat επιτρέπει να 'επισυνάψουμε' μια διαμοιραζόμενη μνήμη σε άλλες διεργασίες. Ο λόγος είναι ότι αρχικά, μια διαμοιραζόμενη μνήμη είναι ορατή μόνο στη διεργασία η οποία τη δημιούργησε. Παράμετροι:

- shm_id: πρόκειται για το ID της μνήμης το οποίο επιστρέφει η shmget.
- shm_addr: είναι η διεύθυνση της διαμοιραζόμενης μνήμης.
- shmflg: είναι ένα σύνολο από bitwise flags. Δύο πιθανές τιμές είναι οι SHM_RND που ελέγχει τη διεύθυνση στην οποία έχει 'επισυναφθεί' η διαμοιραζόμενη μνήμη και η SHM_RDONLY που κάνει τη διαμοιραζόμενη μνήμη να είναι μόνο για ανάγνωση.

Η επιτυχής κλήση της συνάρτησης θα επιστρέψει ένα δείκτη στο πρώτο byte της διαμοιραζόμενης μνήμης ενώ μια αποτυχημένη κλήση επιστρέφει το -1.

Συνάρτηση shmdt

Η συνάρτηση shmdt 'από-επισυνάπτει' τη διαμοιραζόμενη μνήμη από μια διεργασία. Παίρνει ένα δείκτη προς τη διεύθυνση που έχει επιστραφεί από τη συνάρτηση shmat. Σε περίπτωση επιτυχίας η συνάρτηση επιστρέφει 0 ενώ σε αποτυχία επιστρέφει το -1. Θα πρέπει να σημειωθεί ότι η 'από-επισύναψη' της διαμοιραζόμενης μνήμης δεν τη διαγράφει κιόλας αλλά την καθιστά μη προσβάσιμη από την τρέχουσα διεργασία.

Συνάρτηση shmctl

Η συνάρτηση shmctl μας επιτρέπει να εκτελέσουμε συγκεκριμένες εντολές πάνω στη διαμοιραζόμενη μνήμη. Παράμετροι:

- shm_id: πρόκειται για το ID της διεργασίας.
- command: πρόκειται για την εντολή που θα εκτελεστεί πάνω στη διαμοιραζόμενη μνήμη. Οι εντολές παρουσιάζονται στον ακόλουθο πίνακα:

Command	Description
IPC_STAT	Sets the data in the shmid_ds structure to reflect the values associated with the shared memory.
IPC_SET	Sets the values associated with the shared memory to those provided in the shmid_ds data structure, if the process has permission to do so.
IPC_RMID	Deletes the shared memory segment.

- buf:πρόκειται για ένα δείκτη προς μια δομή που περιέχει τα modes και τις άδειες για τη διαμοιραζόμενη μνήμη.

```
struct shmid_ds {
    uid_t shm_perm.uid;
    uid_t shm_perm.gid;
    mode_t shm_perm.mode;
}
```

Σε περίπτωση επιτυχίας η συνάρτηση επιστρέφει 0 ενώ σε αποτυχία επιστρέφει το -1.

Code 3

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main ()
{
  int segment_id;
  char* shared_memory;
  struct shmid_ds shmbuffer;
  int segment_size;
  const int shared_segment_size = 0x6400;

  /* Allocate a shared memory segment.  */
  segment_id = shmget (IPC_PRIVATE, shared_segment_size,
                       IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
  /* Attach the shared memory segment.  */
  shared_memory = (char*) shmat (segment_id, 0, 0);
  printf ("shared memory attached at address %p\n", shared_memory);
  /* Determine the segment's size. */
  shmctl (segment_id, IPC_STAT, &shmbuffer);
  segment_size  =    shmbuffer.shm_segsz;
  printf ("segment size: %d\n", segment_size);
  /* Write a string to the shared memory segment.  */
  sprintf (shared_memory, "Hello, world.");
  /* Detach the shared memory segment.  */
  shmdt (shared_memory);

  /* Reattach the shared memory segment, at a different address.  */
  shared_memory = (char*) shmat (segment_id, (void*) 0x5000000, 0);
  printf ("shared memory reattached at address %p\n", shared_memory);
  /* Print out the string from shared memory.  */
  printf ("%s\n", shared_memory);
  /* Detach the shared memory segment.  */
  shmdt (shared_memory);

  /* Deallocate the shared memory segment.  */
  shmctl (segment_id, IPC_RMID, 0);

  return 0;
}
```

Code 4

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
#define SHM_SIZE 1024  /* make it a 1K shared memory segment */

int main(int argc, char *argv[])
{
   key_t key;
   int shmid;
   char *data;
   int mode;

   if (argc > 2) {
      fprintf(stderr, "usage: shmdemo [data_to_write]\n");
      exit(1);
   }

   /* make the key: */
   if ((key = ftok("hello.txt", 'R')) == -1) /*Here the file must exist */
   {
      perror("ftok");
      exit(1);
   }

   /*  create the segment: */
   if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1) {
      perror("shmget");
      exit(1);
   }

   /* attach to the segment to get a pointer to it: */
   data = shmat(shmid, (void *)0, 0);
   if (data == (char *)(-1)) {
      perror("shmat");
      exit(1);
   }

   /* read or modify the segment, based on the command line: */
   if (argc == 2) {
      printf("writing to segment: \"%s\"\n", argv[1]);
      strncpy(data, argv[1], SHM_SIZE);
   } else
      printf("segment contains: \"%s\"\n", data);

   /* detach from the segment: */
   if (shmdt(data) == -1) {
      perror("shmdt");
      exit(1);
   }

   return 0;
}
```

Code 5

Server

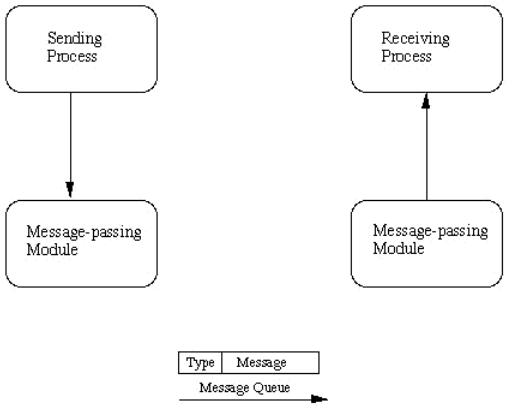
```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

#define MAXSIZE     27

void die(char *s)
{
   perror(s);
   exit(1);
}

int main()
{
   char c;
   int shmid;
   key_t key;
   char *shm, *s;

   key = 5678;

   if ((shmid = shmget(key, MAXSIZE, IPC_CREAT | 0666)) < 0)
      die("shmget");

   if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
      die("shmat");

   /*
    *     * Put some things into the memory for the
    *       other process to read.
    *       */
   s = shm;

   for (c = 'a'; c <= 'z'; c++)
      *s++ = c;


   /*
    * Wait until the other process
    * changes the first character of our memory
    * to '*', indicating that it has read what
    * we put there.
    */
   while (*shm != '*')
      sleep(1);

   exit(0);
}
```

Client

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#define MAXSIZE     27

void die(char *s)
{
   perror(s);
   exit(1);
}

int main()
{
   int shmid;
   key_t key;
   char *shm, *s;

   key = 5678;

   if ((shmid = shmget(key, MAXSIZE, 0666)) < 0)
      die("shmget");

   if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
      die("shmat");

   //Now read what the server put in the memory.
   for (s = shm; *s != '\0'; s++)
      putchar(*s);
   putchar('\n');

   /*
    *Change the first character of the
    *segment to '*', indicating we have read
    *the segment.
    */
   *shm = '*';

   exit(0);
}
```

Ουρές Μηνυμάτων

Οι ουρές μηνυμάτων μοιάζουν με τις σωληνώσεις αλλά δεν έχουν την πολυπλοκότητα τους όσον αφορά στο άνοιγμα και στο κλείσιμο των σωληνώσεων. Οι ουρές μηνυμάτων αποτελούν ένα αποδοτικό τρόπο για να περάσουμε μηνύματα από μια διεργασία σε κάποια άλλη.



Οι συναρτήσεις που χρησιμοποιούνται είναι οι ακόλουθες:

```
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
int msgget(key_t key, int msgflg);
int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);
int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);
```

Συνάρτηση msgget

Με τη συνάρτηση msgget δημιουργούμε και προσπελαύνουμε μια ουρά μηνυμάτων. Παράμετροι:

- key: πρόκειται για το ID της ουράς.
- msgflg: πρόκειται για εννέα (9) παραμέτρους αδειών. Το bit IPC_CREAT πρέπει να ORποιηθεί με τις άδειες χρήσης της νέας ουράς μηνυμάτων.

Σε περίπτωση επιτυχίας η συνάρτηση επιστρέφει ένα θετικό αριθμό ενώ σε αποτυχία επιστρέφει το -1.

Συνάρτηση msgsnd

Η συνάρτηση msgsnd μας επιτρέπει να προσθέσουμε ένα μήνυμα στην ουρά. Παράμετροι:

- msqid: πρόκειται για το ID της ουράς.
- msg_ptr: είναι ένας δείκτης στο μήνυμα που θα σταλεί.
- msg_sz: είναι το μέγεθος του μηνύματος.
- msgflg: υποδεικνύει το τι πρόκειται να συμβεί όταν η ουρά είναι γεμάτη ή έχουμε φτάσει το μέγιστο πλήθος μηνυμάτων όπως καθορίζεται από το σύστημα. Το flag IPC_NOWAIT υποδηλώνει πως στις προ-αναφερόμενες περιπτώσεις η συνάρτηση θα επιστρέψει άμεσα με μήνυμα σφάλματος όταν έχει καθοριστεί. Όταν το IPC_NOWAIT δεν έχει καθοριστεί, η συνάρτηση θα σταματήσει τη διαδικασία αποστολής και θα αναμείνει να υπάρξει χώρος στη ουρά ώστε να προωθήσει το μήνυμα.

Σε περίπτωση επιτυχίας η συνάρτηση επιστρέφει 0 ενώ σε αποτυχία επιστρέφει το -1. Με την επιτυχία, ένα αντίγραφο του μηνύματος θα τοποθετηθεί στην ουρά.

Συνάρτηση msgrcv

Η συνάρτηση msgrcv επιτρέπει τη λήψη ενός μηνύματος από την ουρά. Παράμετροι:

- msqid: πρόκειται για το ID της ουράς.
- msg_ptr: είναι ένας δείκτης στο μήνυμα που θα ληφθεί.

- msg_sz: είναι το μέγεθος του μηνύματος.
- msgtype: πρόκειται για ένα long που επιτρέπει την υλοποίηση μιας πολιτικής λήψης. Αν έχει την τιμή 0 τότε το πρώτο διαθέσιμο μήνυμα στην ουρά θα ληφθεί. Αν έχει μια τιμή μεγαλύτερη από το 0, τότε θα ληφθεί το πρώτο μήνυμα με το συγκεκριμένο τύπο μηνύματος. Αν είναι μικρότερο από το 0, τότε το πρώτο μήνυμα με τύπο ίδιο ή μικρότερο από την απόλυτη τιμή του long θα ληφθεί.
- msgflg: υποδεικνύει το τι πρόκειται να συμβεί όταν η ουρά δεν περιέχει μηνύματα του τύπου που επιθυμούμε. Το flag IPC_NOWAIT υποδηλώνει πως αν δεν υπάρχει μήνυμα με το συγκεκριμένο τύπο, τότε η συνάρτηση θα επιστρέψει άμεσα με μήνυμα σφάλματος (-1) όταν έχει καθοριστεί. Όταν το IPC_NOWAIT δεν έχει καθοριστεί, η συνάρτηση θα αναμείνει να υπάρξει ανάλογο μήνυμα στην ουρά ώστε να το λάβει στη συνέχεια.

Συνάρτηση msgctl

Η συνάρτηση msgctl μας επιτρέπει να εκτελέσουμε συγκεκριμένες εντολές πάνω στην ουρά μηνυμάτων. Παράμετροι:

- msgid: πρόκειται για το ID της ουράς (η τιμή επιστροφής της msgget).
- command: πρόκειται για την εντολή που θα εκτελεστεί πάνω στην ουρά. Οι εντολές παρουσιάζονται στον ακόλουθο πίνακα:

Command	Description
IPC_STAT	Sets the data in the msqid_ds structure to reflect the values associated with the message queue.
IPC_SET	If the process has permission to do so, this sets the values associated with the message queue to those provided in the msqid_ds data structure.
IPC_RMID	Deletes the message queue.

- buf: πρόκειται για ένα δείκτη προς μια δομή που περιέχει τα modes και τις άδειες για την ουρά.

```
struct msqid_ds {
    uid_t  msg_perm.uid;
    uid_t  msg_perm.gid
    mode_t msg_perm.mode;
}
```

Σε περίπτωση επιτυχίας η συνάρτηση επιστρέφει 0 ενώ σε αποτυχία επιστρέφει το -1.

Code 6

Server

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <ctype.h>
#include <fcntl.h>

#define SERVER 1L
typedef struct {
   long    msg_to;
```

```
   long    msg_fm;
   char    buffer[BUFSIZ];
} MESSAGE;

int mid;
key_t key;
struct msqid_ds buf;
MESSAGE msg;

int main(int argc, char** argv) {

   //Creating a message queue
   key = ftok(".", 'z');
   if((mid = msgget(key, IPC_CREAT | 0660))<0){
      printf("Error Creating Message Queue\n");
      exit(-1);
   }

   //Display Message Queue and Server ID
   printf("Message Queue ID: %d\n", mid);
   printf("Server ID: %ld\n", (long)getpid());

   //Receiving message from client, throws and error if input is invalid
   if(msgrcv(mid, &msg, sizeof(msg.buffer), SERVER, 0)<0){
      perror("msgrcv");
      exit(-1);
   }

   //Server displays received message
   printf("SERVER receives:\n");
   printf("%s\n", msg.buffer);

   //Aquiring Cliend PID to message return
   long client = msg.msg_fm;

   //convert all lowercase characters to uppercase
   int i = 0 ;
   while(msg.buffer[i] != '\0'){
      msg.buffer[i] = toupper(msg.buffer[i]);
      i++;
   }

   //prep return message
   msg.msg_fm = SERVER;
   msg.msg_to = client;

   //send converting message back to client, throws and error if input is invalid
   if(msgsnd(mid, (struct MESSAGE*)&msg, sizeof(msg.buffer), 0)==-1){
      perror("msgsnd");
      exit(-1);
   }

   //server exits
   return (EXIT_SUCCESS);
```

```
}
```

Client

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <ctype.h>
#include <fcntl.h>

#define SERVER 1L
typedef struct {
   long    msg_to;
   long    msg_fm;
   char    buffer[BUFSIZ];
} MESSAGE;

int mid;
key_t key;
struct msqid_ds buf;
MESSAGE msg;
FILE *inFile;

int main(int argc, char** argv) {

   //Aquiring Message Queue ID
   key = ftok(".", 'z');
   mid = msgget(key, 0);

   //Display Message Queue and Client ID
   printf("Message Queue ID: %d\n", mid);
   printf("Client ID: %ld\n", (long)getpid());

   //Opeining input file, throw an error if invalid file
   inFile = fopen(argv[1], "r");
   if(inFile == NULL){
      printf("Unable to open File = %s\n", argv[1]);
      return 1;
   }

   //Copy input characters into msg.buffer, loops breaks when EOF is reached
   int i = 0;
   while(1){
      msg.buffer[i] = fgetc(inFile);
      if(msg.buffer[i]==EOF){
         msg.buffer[i] = '\0';
         break;
      }
      i++;
   }
```

```
    //Displaying message before conversion of server
    printf("Message before conversion:\n");
    printf("%s\n", msg.buffer);

    //Getting Client PID and preparing message to message queue
    long iD = (long)getpid();
    msg.msg_to = SERVER;
    msg.msg_fm = (long)getpid();

    //Send message to Message Queue for Server, throws and error for invalid input
    if(msgsnd(mid, &msg, sizeof(msg.buffer), 0)==-1){
       perror("msgsnd");
       exit(-1);
    }

    //Client waits for response from Server, throws an error if invalid input
    if(msgrcv(mid, &msg, sizeof(msg), iD, 0)<0){
       perror("msgrcv");
       exit(-1);
    }

    //Display new converting message.
    printf("Message after conversion\n");
    printf("%s\n", msg.buffer);

    //Removing message queue
    msgctl(mid, IPC_RMID, (struct msqid_ds *) 0);

    //Client exits
    return (EXIT_SUCCESS);
}
```

Σχετικές Εντολές

- **ipcs –s**

  Εμφανίζει τη κατάσταση των σημαφόρων του συστήματος.

```
------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x4d00df1a 768          rick      666        1
```

- **ipcrm -s <id>**

  Επιτρέπει τη διαγραφή των σημαφόρων.

  Παράδειγμα: ipcrm –s 768

- **ipcrm sem <id>**

  Επιτρέπει τη διαγραφή των σημαφόρων.

  Παράδειγμα: ipcrm sem 768

- **ipcs –m**

Εμφανίζει τη κατάσταση των διαμοιραζόμενων μνημών του συστήματος.

```
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 384        rick       666        4096       2          dest
```

- **ipcrm –m <id>**

  Διαγράφει μια διαμοιραζόμενη μνήμη.

- **ipcs –q**

  Εμφανίζει τη κατάσταση των ουρών μηνυμάτων του συστήματος.

```
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x000004d2 3384       rick       666        2048            2
```

- **ipcrm –q <id>**

  Διαγράφει μια ουρά μηνυμάτων.