ΕΡΓΑΣΤΗΡΙΟ 9

Διαδιεργασιακή Επικοινωνία

Να πληκτρολογηθεί και να εκτελεστεί ο κώδικας των επόμενων παραδειγμάτων που επιδεικνύουν τους διάφορους τρόπους επικοινωνίας μεταξύ διεργασιών.

Παράδειγμα 1. Επικοινωνία διεργασιών με κλείδωμα κοινόχρηστου αρχείου (αρχεία producer.c και consumer.c).

Αρχείο producer.c

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

Το κοινόχρηστο αρχείο που θα χρησιμοποιηθεί έχει όνομα data.dat ενώ το μήνυμα που θα αποθηκευτεί στο αρχείο ορίζεται ως η σταθερή συμβολοσειρά DataString.

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

Σε αυτή την εφαρμογή, οι δύο διεργασίες εκ των οποίων η πρώτη αποτελεί μία διεργασία συγγραφέα ενώ η δεύτερη, μία διεργασία αναγνώστη, επικοινωνούν μεταξύ τους, δηλαδή ανταλλάσσουν ένα μήνυμα χρησιμοποιώντας ένα κοινόχρηστο αρχείο. Με άλλα λόγια, η διεργασία συγγραφέας ανοίγει το αρχείο, αποθηκεύει σε αυτό το μήνυμα και κλείνει το αρχείο, ενώ στη συνέχεια η διεργασία αναγνώστης, ανοίγει το αρχείο, διαβάζει το μήνυμα που της έστειλε η διεργασία συγγραφέας και κλείνει το αρχείο. Προκειμένου να αποτρέψουμε την ταυτόχρονη προσπέλαση του αρχείου από τις δύο διεργασίες χρησιμοποιείται ένας μηχανισμός κλειδώματος έτσι ώστε να διασφαλίσουμε ότι μόνο μία διεργασία μπορεί να ανοίξει το αρχείο σε κάθε χρονική στιγμή.

```
int main() {
  struct flock lock;
```

Το πιο σημαντικό κομμάτι του κώδικα σε αυτή την περίπτωση είναι η αρχικοποίηση της δομής lock. Ορίζοντας ως l_type την τιμή F_WRLCK κλειδώνουμε το αρχείο για αποκλειστική χρήση (exclusive lock) προκειμένου να πραγματοποιηθεί διαδικασία εγγραφής, ενώ τα υπόλοιπα ορίσματα κλειδώνουν ολόκληρο το αρχείο, από το πρώτο έως το τελευταίο byte (EOF). Το τελευταίο πεδίο είναι το PID της διεργασίας που κλειδώνει το αρχείο.

```
  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 */
```

Η διεργασία – συγγραφέας ανοίγει το αρχείο data.dat για ανάγνωση / εγγραφή (O_RDWR), το δημιουργεί εάν δεν υπάρχει (O_CREAT) και με default mode 0666 (rw-rw-rw-)

```
  if ((fd = open(FileName, O_RDWR | O_CREAT, 0666)) < 0)  {
      perror("open failed..."); exit(-1); }
```

Το αρχείο τίθεται σε κατάσταση exclusive lock περνώντας ως όρισμα στην fcntl τη δομή lock.

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

```
  else {
```

Η διεργασία συγγραφέας αποθηκεύει στο αρχείο το περιεχόμενο της συμβολοσειράς DataString

```
   write(fd, DataString, strlen(DataString)); /* populate data file */
   fprintf(stderr, "Process %d has written to data file...\n", lock.l_pid); }
```

Στο τέλος της διαδικασίας το αρχείο ξεκλειδώνει έτσι ώστε να είναι προσπελάσιμο από άλλες διεργασίες και τελικά κλείνει αφού δεν χρειάζεται πλέον.

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

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

Αρχείο consumer.c

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#define FileName "data.dat"
```

Όπως αναφέραμε προηγουμένως, η διεργασία αναγνώστης, ανοίγει το αρχείο, διαβάζει το μήνυμα που της έστειλε η διεργασία συγγραφέας και κλείνει το αρχείο.

```
int main() {
  struct flock lock;
```

Η αρχικοποίηση της δομής lock γίνεται ακριβώς όπως και πριν

```
  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 */
```

Εδώ το αρχείο ανοίγει μόνο για ανάγνωση.

```
  if ((fd = open(FileName, O_RDONLY)) < 0) {
      perror("open to read failed..."); exit(-1); }
```

Στην προκειμένη περίπτωση πριν επιχειρήσουμε να διαβάσουμε από το αρχείο θα πρέπει να ελέγξουμε εάν αυτό είναι κλειδωμένο καλώντας την συνάρτηση fcntl με όρισμα *F_GETLK*. *Εάν η επιστρεφόμενη τιμή της συνάρτησης δεν είναι η F_UNLCK αυτό σημαίνει πως κάποια άλλη διεργασία έχει κλειδώσει το αρχείο προκειμένου να το χρησιμοποιήσει με αποκλειστικό τρόπο. Επομένως δεν είναι δυνατή η προσπέλαση του από τη διεργασία αναγνώστη, η οποία για το λόγο αυτό εκτυπώνει ένα μήνυμα λάθους και τερματίζει τη λειτουργία της.*

```
  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); }
```

Στη συνέχεια, η διεργασία αναγνώστης κλειδώνει με τη σειρά της το αρχείο προκειμένου κατά τη διάρκεια της διαδικασίας ανάγνωσης του περιεχομένου του να μην μπορεί να γράψει σε αυτό καμία άλλη διεργασία.

```
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); }
```

Στη συνέχεια διαβάζει το μήνυμα που είναι αποθηκευμένο στο αρχείο ένα χαρακτήρα κάθε φορά και το εκτυπώνει στην προεπιλεγμένη της έξοδο που είναι η οθόνη.

```
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 */
```

Στο τέλος της διαδικασίας το αρχείο ξεκλειδώνει έτσι ώστε να είναι προσπελάσιμο από άλλες διεργασίες και τελικά κλείνει αφού δεν χρειάζεται πλέον.

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

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

Παράδειγμα εξόδου της διεργασίας, ακολουθεί στη συνέχεια. Η κάθε διεργασία εκτελείται στο δικό της τερματικό, αν και στην προκειμένη περίπτωση επειδή οι διεργασίες δεν επικοινωνούν μεταξύ τους μέσω της ανταλλαγής μηνυμάτων, μπορούν να εκτελεστούν η μία μετά την άλλη στο ίδιο τερματικό. Στην περίπτωση αυτή, η πρώτη διεργασία θα ανοίξει το αρχείο, θα γράψει το μήνυμα της και θα το κλείσει ολοκληρώνοντας τη λειτουργία της, ενώ στη συνέχεια εκτελείται η δεύτερη διεργασία, η οποία ανοίγει το αρχείο, διαβάζει το μήνυμα, το εκτυπώνει στην οθόνη της και τέλος κλείνει το αρχείο. Με τον τρόπο αυτό οι δύο διεργασίες επικοινωνούν μεταξύ τους.

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

Παράδειγμα 2. Επικοινωνία διεργασιών με χρήση κοινόχρηστης μνήμης (αρχεία memwriter.c και memreader.c).

Η εφαρμογή memwriter γράφει ένα ενδεικτικό μήνυμα (*This is the way the world ends*) στην κοινόχρηστη μνήμη χρησιμοποιώντας βοηθητικό αρχείο και αναμένει για 12 δευτερόλεπτα έτσι ώστε να δώσει τη δυνατότητα στην εφαρμογή memreader να διαβάσει τα περιεχόμενα από την κοινόχρηστη μνήμη.

Αρχείο memwriter.c

```
#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>
#include "shmem.h"

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

int main() {

Σε αυτή την εφαρμογή, η επικοινωνία ανάμεσα στις δύο διεργασίες στηρίζεται στη χρήση ενός βοηθητικού αρχείου το οποίο δημιουργείται από την εφαρμογή memwriter. Η εντολή shm_open δέχεται παρόμοια ορίσματα με την open και στην προκειμένη περίπτωση ανοίγει το αρχείο *shMemEx* το οποίο *δημιουργεί* (O_CREAT) για διαδικασίες *ανάγνωσης / εγγραφής* (O_RDWR) με δικαιώματα 0644 (rw-r—r--). Είναι σημαντικό να γίνει κατανοητό πως το εν λόγω αρχείο υφίσταται μόνο για το χρονικό διάστημα εκτέλεσης της εν λόγω διεργασίας, αφού με τον τερματισμό της, το αρχείο διαγράφεται από το δίσκο. Αυτό το αρχείο μπορεί να βρεθεί στον κατάλογο /dev/shm του συστήματος αρχείων.

int fd = shm_open ("/shMemEx", O_RDWR | O_CREAT, 0644);

Εάν για κάποιο λόγο το αρχείο δεν μπορεί να δημιουργηθεί η εφαρμογή δεν γίνεται να εκτελεστεί και ως εκ τούτου τερματίζει εκτυπώνοντας το κατάλληλο μήνυμα σφάλματος.

if (fd < 0) { perror ("Can't open shared mem segment..."); exit (-1); }

Εάν το αρχείο δημιουργηθεί με επιτυχία καλείται η συνάρτηση ftruncate για να ορίσει το μέγεθος του αρχείου το οποίο τίθεται στην τιμή ByteSize (το μέγιστο μήκος του μηνύματος που μπορεί να ανταλλαγεί ανάμεσα στις διεργασίες).

ftruncate(fd, ByteSize);

Σε αυτό το σημείο καλείται η συνάρτηση mmap για να δημιουργήσει μία νέα απεικόνιση στον εικονικό χώρο διευθύνσεων της διεργασίας. Αν και μπορούμε να ορίσουμε ως πρώτο όρισμα σε αυτή τη συνάρτηση κάποια διεύθυνση, ωστόσο, αφήνουμε αυτό το όρισμα στην τιμή NULL προκειμένου να αποφασίσει ο πυρήνας σχετικά με την περιοχή μνήμης στην οποία θα τοποθετηθεί η απεικόνιση. Ως μήκος για αυτή την απεικόνιση ορίζουμε την τιμή ByteSize που είναι το μέγιστο μήκος του μηνύματος που θα ανταλλαγεί ανάμεσα στις δύο διεργασίες. Στη συνέχεια χρησιμοποιούμε τα flags PROT_READ και PROT_WRITE ορίζοντας για αυτή την περιοχή δικαιώματα ανάγνωσης και εγγραφής καθώς και το flag MAP_SHARED που καθιστά κοινόχρηστη αυτή την απεικόνιση (κάτι που είναι αναγκαίο προκειμένου στη διαδικασία να χρησιμοποιηθεί το βοηθητικό αρχείο μέσω του οποίου οι δύο διεργασίες θα χρησιμοποιήσουν από κοινού αυτή την περιοχή μνήμης). Το επόμενο όρισμα είναι ο περιγραφέας αρχείου fd για αυτό το βοηθητικό αρχείο, ο οποίος έχει επιστραφεί από τη συνάρτηση shm_open ενώ το τελευταίο όρισμα που εκφράζει την απόσταση μέσα στο αρχείο στο οποίο θα τοποθετηθούν τα κοινόχρηστα δεδομένα τίθεται στη μηδενική τιμή, έτσι ώστε αυτά να τοποθετηθούν στην αρχή του αρχείου. Εάν η απεικόνιση δεν μπορεί να δημιουργηθεί για κάποιο λόγο, η εκτέλεση της εφαρμογής δεν είνα δυνατή, οπότε τερματίζει εκτυπώνοντας το κατάλληλο μήνυμα σφάλματος.

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); }

Στο επόμενο βήμα ορίζουμε έναν σημαφόρο χρησιμοποιώντας τη συνάρτηση sem_open. Επειδή ο σημαφόρος δημιουργείται από τη συνάρτηση αφού χρησιμοποιούμε το flag O_CREAT θα πρέπει να ορίσουμε τα permissions (που τα θέτουμε στην τιμή 0644) και την αρχική τιμή του σημαφόρου που είναι η τιμή 0. Εάν ο σημαφόρος δεν δημιουργηθεί, η εκτέλεση της εφαρμογής δεν είνα δυνατή, οπότε και πάλι τερματίζει εκτυπώνοντας το κατάλληλο μήνυμα σφάλματος.

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

Εάν όλες οι παραπάνω διαδικασίες πραγματοποιηθούν χωρίς πρόβλημα οι δύο διαδικασίες μπορούν πλέον να επικοινωνήσουν μεταξύ τους. Σε αυτό λοιπόν το σημείο καλούμε τη συνάρτηση strcpy προκειμένου να αντιγράψουμε το περιεχόμενο της συμβολοσειράς που περιέχει το μήνυμα στην κοινόχρηστη περιοχή μνήμης.

strcpy(memptr, MemContents);

Στη συνέχεια αυξάνουμε την τιμή του σημαιοφόρου κατά μία μονάδα (unlock) προκειμένου να δώσουμε τη δυνατότητα στην άλλη διεργασία να διαβάσει το μήνυμα (εάν αυτή η συνάρτηση δεν εκτελεστεί σωστά, ξανά η διεργασία τερματίζεται εκτυπώνοντας μήνυμα σφάλματος) και αμέσως μετά η διεργασία αναστέλλεται για 12 δευτερόλεπτα (ο αριθμός είναι ενδεικτικός : μπορείτε να βάλετε οποίο νούμερο θέλετε) έτσι ώστε η άλλη διεργασία να προλάβει να εκτελεστεί και να διαβάσει το μήνυμα από το κοινόχρηστο αρχείο.

if (sem_post (semptr) < 0) { perror ("sem_post"); exit (-3); }
sleep(12); / give reader a chance */*

Μετά την παρέλευση των 12 δευτερολέπτων κατά τη διάρκεια των οποίων (ευελπιστούμε πως) η άλλη διεργασία προσπέλασε το αρχείο και διάβασε το κοινόχρηστο μήνυμα, η διαδικασία ολοκληρώνεται καταργώντας την απεικόνιση μνήμης και κλείνοντας τα εμπλεκόμενα αρχεία και τις δομές που χρησιμοποιήθηκαν (σημαφόροι).

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

return 0; }
```

Η εφαρμογή memreader εντός 12 δευτερολέπτων από την έναρξη της memwriter, διαβάζει από την κοινόχρηστη μνήμη το περιεχόμενό της και το εκτυπώνει στην οθόνη.

Αρχείο memreader.c

```
#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>
#include "shmem.h"

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

Η διεργασία memreader σχεδόν σε όλα λειτουργεί με τον ίδιο ακριβώς τρόπο και ως εκ τούτου ισχύουν όλα όσα αναφέραμε στην διεργασία memwriter. Η μόνη διαφορά είναι ότι στη συνάρτηση shm_open δεν χρησιμοποιούμε το flag O_CREAT αλλά μόνο το O_RDWR αφού το αρχείο έχει ήδη δημιουργηθεί από τη διεργασία memwriter. Τα υπόλοιπα ισχύουν ως έχουν.

```
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); }
```

Ωστόσο, υπάρχει διαφοροποίηση στον τρόπο χρήσης του σημαφόρου. Θυμηθείτε πως για να είναι δυνατή η ανάγνωση του αρχείου από τη διεργασία memreader, η διεργασία memwriter θα πρέπει να αυξήσει την τιμή του σημαφόρου κατά μία μονάδα καλώντας την συνάρτηση sem_post έτσι ώστε να ξεκλειδώσει το αρχείο για να χρησιμοποιηθεί από τη διεργασία memreader. Εάν η τιμή του σημαφόρου δεν είναι διαφορετική του μηδενός (και ειδικότερα, μεγαλύτερη του μηδενός) η διεργασία memreader δεν μπορεί να προσπελάσει το αρχείο. Εκείνο που κάνει είναι να καλεί τη συνάρτηση sem_wait η οποία αναμένει να λάβει ο σημαφόρος θετική τιμή και επιστρέφει μηδενική τιμή όταν αυτό συμβεί. Όταν λοιπόν ο σημαφόρος αυξηθεί κατά μία μονάδα και το αρχείο έχει ξεκλειδώσει, τότε η διεργασία memreader διαβάζει ένα προς ένα τα bytes από την περιοχή μνήμης που υποδηλώνεται από το δείκτη memprt που επέστρεψε η συνάρτηση mmap και μέσω μίας επαναληπτικής διαδικασίας που εκτελείται τόσες φορές όση είναι και η τιμή του ByteSize εκτυπώνει τους χαρακτήρες στην προεπιλεγμένη έξοδο που είναι η οθόνη.

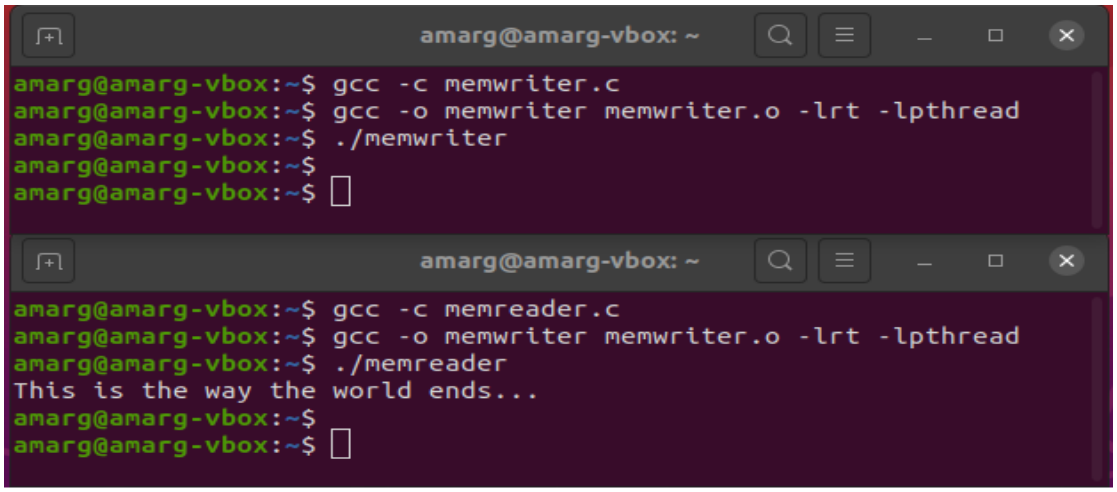
```
  if (!sem_wait(semptr)) { /* wait until semaphore != 0 */
     int i;
     for (i = 0; i < strlen (MemContents); i++)
```

```
        write(STDOUT_FILENO, memptr + i, 1);
    sem_post(semptr); }
```

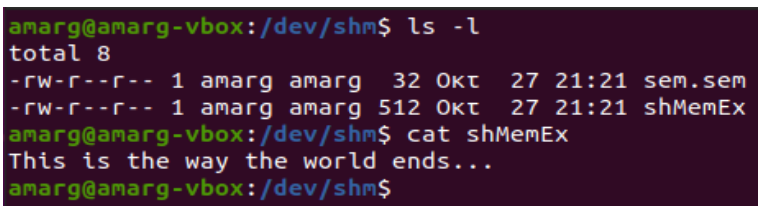
Η διαδικασία ολοκληρώνεται καταργώντας την απεικόνιση μνήμης και κλείνοντας τα εμπλεκόμενα αρχεία και τις δομές που χρησιμοποιήθηκαν (σημαφόροι).

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

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.



Παράδειγμα 3. Επικοινωνία διεργασιών με χρήση ουράς μηνυμάτων (αρχείο mqExample.c).

```
#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
```

Σε αυτή την εφαρμογή, επιδεικνύεται η χρήση μιας ουράς μηνυμάτων (message queue) η οποία υλοποιείται ως ένα αρχείο του σκληρού δίσκου και η οποία έχει τη δυνατότητα να φιλοξενήσει ένα συγκεκριμένο αριθμό μηνυμάτων. Για λόγους απλότητας, δεν χρησιμοποιούμε δύο διεργασίες, αλλά μόνο μία διεργασία η οποία εκτελείται δύο φορές: την πρώτη φορά αποθηκεύει κάποια μηνύματα στην ουρά μηνυμάτων, ενώ τη δεύτερη φορά διαβάζει τα μηνύματα που είχαν αποθηκευτεί κατά την πρώτη εκτέλεση και γράφει άλλα 10 μηνύματα. Βέβαια, σε μία πραγματική εφαρμογή, αυτή η διαδικασία θα μπορούσε να υλοποιηθεί χρησιμοποιώντας δύο διεργασίες, με τη δεύτερη διεργασία να ανοίγει το αρχείο της ουράς μηνυμάτων που έχει δημιουργήσει η πρώτη διεργασία και να διαβάζει τα μηνύματα που έχουν αποθηκευτεί από αυτή και

γράφοντας με τη σειρά της τα δικά της μηνύματα, τα οποία στη συνέχεια θα μπορούσαν να διαβαστούν από την πρώτη διεργασία, κ.ο.κ. Ωστόσο εδώ, για λόγους απλότητας, όπως σχολιάσαμε, αυτή η διαδικασία πραγματοποιείται από μία και μοναδική διεργασία η οποία εκτελείται δύο φορές.

```
void handler (int sig_num) { printf ("Received sig %d.\n", sig_num); }
```
Ο χειριστής του σήματος SIGUSR1

```
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;
```

Αρχικά δημιουργούμε μία νέα ουρά μηνυμάτων καλώντας τη συνάρτηση mq_open. Τα μηνύματα αυτής της ουράς αποθηκεύονται στο αρχείο του δίσκου mqueue1 το οποίο δημιουργείται στον κατάλογο /dev/mqueue. Το αρχείο δημιουργείται για ανάγνωση και εγγραφή (O_RDWR | O_CREAT) με δικαιώματα 0664 ενώ επειδή το τελευταίο όρισμα της συνάρτησης είναι NULL χρησιμοποιούνται για την ουρά προεπιλεγμένες ιδιότητες.

```
  mqdes = mq_open ("/mqueue1", O_RDWR | O_CREAT, 0664, NULL);
```

Μετά τη δημιουργία της ουράς καλούμε τη συνάρτηση mq_getattr για να εκτυπώσουμε τις τιμές των (προεπιλεγμένων) ιδιοτήτων της και πιο συγκεκριμένα το μέγιστο πλήθος των μηνυμάτων που μπορεί να φιλοξενήσει, το μήκος του κάθε μηνύματος καθώς και το πλήθος των μηνυμάτων που βρίσκονται στην ώρα αυτή τη χρονική στιγμή.

```
  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);
```

Εάν στην ουρά μηνυμάτων υπάρχουν μηνύματα για παραλαβή, αυτά παραλαμβάνονται μέσω μίας επαναληπτικής διαδικασίας, η οποία καλεί τη συνάρτηση mq_receive για να παραλάβει ένα μήνυμα σε κάθε κύκλο επανάληψης. Το όρισμα *O_NONBLOCK χρησιμοποιείται έτσι ώστε η συνάρτηση να μην μπλοκάρει εάν δεν βρει μηνύματα προς παραλαβή αλλά αντίθετα, να επιστρέψει αμέσως, με κωδικό σφάλματος EAGAIN. Αυτός είναι ο μοναδικός κωδικός σφάλματος ο οποίος επιτρέπεται να παραληφθεί, ενώ αν επιστραφεί οποιοσδήποτε άλλος κωδικός σφάλματος, αυτό σημαίνει πως έλαβε χώρα κάποιο άλλο σφάλμα και η εφαρμογή τερματίζει τη λειτουργία της. Λαμβάνοντας υπόψη πως η εφαρμογή θα εκτελεστεί δύο φορές και έτσι ώστε την πρώτη φορά μόνο να αποστείλει μηνύματα στην ουρά μηνυμάτων, ενώ τη δεύτερη φορά να διαβάσει τα μηνύματα που αποθήκευσε την πρώτη φορά, είναι προφανές πως ο κώδικας που βρίσκεται στο block if (attr.mq_curmsgs != 0) θα εκτελεστεί μόνο τη δεύτερη , διότι μόνο τότε θα βρει μηνύματα για να παραλάβει.*

```
  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)
        printf ("Received a message with priority %d.\n", prio);
    if (errno != EAGAIN) { perror ("mq_receive()"); exit (EXIT_FAILURE); }
    mq_setattr (mqdes, &old_attr, 0); }
```

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) == -1) {
    if (errno == EBUSY)
      printf ("Another process has registered for notification.\n");
      exit (EXIT_FAILURE); }
```

Ανεξάρτητα από το εάν θα λάβει χώρα η διαδικασία ανάγνωσης των μηνυμάτων της ουράς ή όχι (αυτό εξαρτάται από το εάν η εφαρμογή εκτελείται για πρώτη ή για δεύτερη φορά), σε αυτό το σημείο η εφαρμογή μέσω μιας επαναληπτικής διαδικασίας και χρησιμοποιώντας τη συνάρτηση mq_send αποστέλλει στην ουρά μηνυμάτων δέκα νέα μηνύματα. Προκειμένου να επιδείξουμε το γεγονός πως οι ουρές μηνυμάτων παρά το γεγονός πως αποτελούν δομές FIFO υποστηρίζουν τιμές προτεραιότητας και κατά συνέπεια το μήνυμα που διαβάζεται πρώτο είναι το παλαιότερο μήνυμα (δηλαδή αυτό που στάλθηκε πρώτο) αλλά με τη μέγιστη τιμή προτεραιότητας, τα δέκα μηνύματα που αποθηκεύονται στο αρχείο της ουράς, έχουν τιμές προτεραιότητας οι οποίες αυξάνονται κατά 5.

```
for (i= 0; i < attr.mq_maxmsg; i++) {
    printf ("Writing a message with priority %d.\n", prio);
    if (mq_send (mqdes, message, strlen(message)+1, prio) == -1) perror ("mq_send()");
    prio += 5;}
```

Μετά την ολοκλήρωση της διαδικασίας αποστολής, η διεργασία τερματίζεται κλείνοντας τον περιγραφέα του αρχείου της ουράς μηνυμάτων.

```
mq_close (mqdes);
return (0); }
```

Παράδειγμα εξόδου της εφαρμογής, ακολουθεί στη συνέχεια. 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.

```
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:~$
```

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

```
amarg@amarg-vbox:/dev/mqueue$ ls -l
total 0
-rw-rw-r-- 1 amarg amarg 80 Νοε   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$
```