Προγραμματισμός συστημάτων UNIX/POSIX

Προχωρημένη διαδιεργασιακή επικοινωνία:

επώνυμοι αγωγοί (FIFOs)

ουρές μηνυμάτων (message queues)

κοινόχρηστη μνήμη (shared memory)

σήματα (signals)

MYY502

Ανάγκη

- Ότι είδαμε μέχρι τώρα μπορεί να χρησιμοποιηθεί για επικοινωνία διεργασιών που έχουν σχέση πατέρα – παιδιού
  - Διότι κληρονομούνται οι περιγραφείς αρχείων.
- Είναι περιοριστικό όμως. Μπορούμε να κάνουμε δύο ανεξάρτητες διεργασίες να επικοινωνήσουν;;;
- Απάντηση: Ναι! Και μάλιστα υπάρχουν πολλοί διαφορετικοί μηχανισμοί για το σκοπό αυτό.
  - «Προχωρημένη» διαδιεργασιακή επικοινωνία (interprocess communication – IPC).
  - Δεν υποστηρίζονται πάντα όλες οι διαφορετικές εκδόσεις των μηχανισμών σε όλα τα συστήματα

Μηχανισμοί IPC

- **Επώνυμοι αγωγοί** (named pipes ή FIFOs)
- **Ουρές μηνυμάτων** (message queues)
  - Δύο εκδόσεις: System V (SysV) και POSIX
- **Κοινόχρηστη μνήμη** (shared memory)
  - Δύο εκδόσεις: System V (SysV) και POSIX
- **Υποδοχές** (sockets)
- κλπ.

- Όλοι οι μηχανισμοί αφορούν διεργασίες οι οποίες εκτελούνται στο ίδιο μηχάνημα
- Οι υποδοχές είναι ο μοναδικός μηχανισμός που *επιπρόσθετα επιτρέπει και την επικοινωνία διεργασιών οι οποίες εκτελούνται σε διαφορετικά συστήματα*.

FIFOs

- Πρόκειται για ένα «ειδικό» δυαδικό αρχείο το οποίο μπορούν να το ανοίξουν για διάβασμα ή γράψιμο δύο ανεξάρτητες διεργασίες αρκεί να ξέρουν το όνομά του (διαδρομή).
- *Δημιουργία ενός FIFO*:

```
int mkfifo(char *fullpath, mode_t permissions);
```

  όπου τα permissions είναι οι άδειες, ίδιες με αυτές που δίνονται και κατά τη δημιουργία ενός νέου δυαδικού αρχείου στο τρίτο όρισμα της `open()`.

- Αφού δημιουργηθεί το FIFO από κάποια διεργασία με την `mkfifo()`, οι εμπλεκόμενες διεργασίες (ακόμα και αυτή που το δημιούργησε) **πρέπει να το ανοίξουν με την `open()`**.
- Από κει και μετά, λειτουργούν όπως οι αγωγοί με χρήση των `write` και `read`.
- Κατά την `open()`, *η διεργασία που ανοίγει το FIFO για εγγραφή **μπλοκάρει** μέχρι μία άλλη διεργασία να κάνει `open()` για ανάγνωση και το αντίστροφο*.
  - Όμως, αν απαιτείται, η διεργασία που ανοίγει για ανάγνωση μπορεί κατά την `open()` να δώσει το ειδικό flag `O_NONBLOCK` ώστε να μην μπλοκάρει, π.χ.
    `O_RDONLY | O_NONBLOCK`.

Παράδειγμα με FIFO: πελάτης

Πελάτης–client.c

```
#include "fifo.h"
int main() {
  int readfd, writefd;
  /* Open the FIFOs. We assume the server has already created them */
  if ( (writefd = open(FIFO1, O_WRONLY)) < 0)
    printf("client: can't open write fifo: %s", FIFO1);
  if ( (readfd = open(FIFO2, O_RDONLY)) < 0)
    printf("client: can't open read fifo: %s", FIFO2);

  <... Κώδικας του πελάτη ...>

  close(readfd);
  close(writefd);
  /* Delete the FIFOs, now that we're finished. */
  if (unlink(FIFO1) < 0)
    printf("client: can't unlink %s", FIFO1);
  if (unlink(FIFO2) < 0)
    printf("client: can't unlink %s", FIFO2);
  return 0;
}
```

fifo.h

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"
#define PERMS S_IRUSR | S_IWUSR
```

Παράδειγμα με FIFO: εξυπηρετητής

Εξυπηρετητής – server.c

```
#include "fifo.h"
int main() {
  int readfd, writefd;
  /* Create the FIFOs -- one for reading and one for writing. */
  if (mkfifo(FIFO1, PERMS) < 0)
    printf("can't create fifo: %s", FIFO1);
  if (mkfifo(FIFO2, PERMS) < 0) {
    unlink(FIFO1);
    printf("can't create fifo: %s", FIFO2);
  }
  /* Open the FIFOs – one for reading and one for writing */
  if ( (readfd = open(FIFO1, O_RDONLY)) < 0)                    /* A */
    printf("server: can't open read fifo: %s", FIFO1);
  if ( (writefd = open(FIFO2, O_WRONLY)) < 0)                   /* B */
    printf("server: can't open write fifo: %s", FIFO2);

  <... Κώδικας του εξυπηρετητή ...>

  close(readfd);
  close(writefd);
  return 0;
}
```

Τι θα γίνει αν εναλλάξουμε τις δύο γραμμές A και B ?

fifo.h

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"
#define PERMS S_IRUSR | S_IWUSR
```

SysV IPC

- Οι μηχανισμοί διαδιεργασιακής επικοινωνίας SysV
  a. Απαιτούν ένα **μοναδικό «κλειδί»** (συνήθως ακέραιος αριθμός)
  b. Δεν τους χειριζόμαστε όπως τα δυαδικά αρχεία (δεν διαθέτουν περιγραφέα αρχείου)
  c. Αφήνουν «υπολείμματα» κατά τον τερματισμό μίας εφαρμογής και για αυτό το λόγο πρέπει ο προγραμματιστής να τα «διαγράφει» ρητά, αλλιώς ξεμένουν και γεμίζουν τη μνήμη του συστήματος.

- Για τη δημιουργία του κλειδιού
  - είτε διαλέγουμε έναν «περίεργο» αριθμό που θεωρούμε ότι δεν τον χρησιμοποιούν άλλες διεργασίες
  - είτε, για μεγαλύτερη ασφάλεια, χρησιμοποιούμε τη συνάρτηση `ftok()` η οποία δημιουργεί μοναδικά κλειδιά αν της περάσουμε μια οποιαδήποτε υπάρχουσα διαδρομή αρχείου.

```
#include <sys/ipc.h>
key_t ftok(char *fullpath, int id);
```

  - Η διαδρομή πρέπει να είναι οποιοδήποτε υπάρχον αρχείο ή κατάλογος και ο αριθμός `id` οποιοσδήποτε ακέραιος, π.χ. `x = ftok("/tmp", 1);`
  - Το σύστημα εξασφαλίζει τη μοναδικότητα του κλειδιού.

Ουρές μηνυμάτων (SysV message queues)

- Οι αγωγοί και τα FIFOs είναι μηχανισμοί επικοινωνίας ως μία ροή από bytes.
- Οι ουρές μηνυμάτων είναι μία λίστα από `structs`, καθένα από τα οποία περιέχει ένα διαφορετικό «μήνυμα» από μία διεργασία προς μία άλλη. Τη λίστα τη χειρίζεται το λειτουργικό σύστημα.
- Τα μηνύματα μπορεί να τα διαβάσει μία διεργασία με όποια σειρά θέλει.
- Οι ουρές μηνυμάτων είναι **δικατευθυντήριες** (δηλ. μπορεί να γράψουν και να διαβάσουν και οι δύο διεργασίες).

Κλήσεις για ουρές μηνυμάτων

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

```
int msgget(key_t key, int msgflag);
```

- Δημιουργία και άνοιγμα ουράς μηνυμάτων με κλειδί το `key`. Το `msgflag` είναι συνήθως `IPC_CREAT` (όχι `O_CREAT`!). Επιστρέφει ένα περιγραφέα (id) για την ουρά.

```
int msgsnd(int msgqid, void *ptr, int length, int flag);
```

- Αποστολή-προσθήκη μηνύματος στην ουρά μηνυμάτων. **Ο δείκτης `ptr` πρέπει να δείχνει σε ένα struct το οποίο έχει 2 πεδία**: α) έναν `long int` που αντιπροσωπεύει τον «τύπο» του μηνύματος και β) τα δεδομένα του μηνύματος.

```
int msgrev(int msgqid, void *ptr, int length, long msgtype, int flag);
```

- Παραλαβή-αφαίρεση μηνύματος από την ουρά μηνυμάτων. Παραλαμβάνει μόνο μήνυμα του δεδομένου τύπου (ή όποιο μήνυμα είναι πρώτο στην ουρά αν το `msgtype` είναι 0)

```
int msgctl(int msgqid, int cmd, struct msgqid_ds *buff);
```

- Χρησιμοποιείται για έλεγχο της ουράς (π.χ. διαγραφή της).

Κοινόχρηστη μνήμη (SysV shared memory)

- Εντελώς διαφορετικός μηχανισμός και συνήθως ο ταχύτερος.
- Χοντρικά κάποια διεργασία δεσμεύει μνήμη (κάτι σαν malloc) την οποία όμως την προσπελαύνουν πολλές διεργασίες!
- Όποια διεργασία τροποποιήσει κάτι, οποιαδήποτε άλλη μπορεί να δει την αλλαγή.
- Μπαίνουν θέματα **ταυτοχρονισμού / παραλληλισμού**. Τι γίνεται αν π.χ. δύο ή παραπάνω διεργασίες πάνε να τροποποιήσουν το ίδιο byte???
- Συνήθως συνδυάζονται με σηματοφόρους (**semaphores**) για τον έλεγχο του παραλληλισμού και το συγχρονισμό των διεργασιών.

Κλήσεις για κοινόχρηστη μνήμη

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

```
int shmget(key_t key, size_t size, int msgflag);
```

- Δέσμευση τμήματος μνήμης (κάτι σαν «malloc») με κλειδί το key. Το msgflag είναι συνήθως IPC_CREAT (όχι O_CREAT!). Επιστρέφει έναν περιγραφέα (id) για το τμήμα μνήμης.

```
void *shmat(int shmid, void *ptr, int flag);
```

- Η shmget() δεν επιστρέφει διεύθυνση μνήμης όπως η malloc(). Αυτό το κάνει η shmat() η οποία και «προσαρτεί» τον κοινόχρηστο χώρο μνήμης με το δεδομένο id στην τρέχουσα διεργασία. Το δεύτερο όρισμα είναι συνήθως NULL και το τρίτο 0. Εφόσον όλες οι διεργασίες εκτελέσουν την shmat(), ότι γράφει κάποια μέσα στο χώρο αυτό επηρεάζει όλες.

```
int shmctl(int shmid, int cmd, struct shmid_ds *buff);
```

- Χρησιμοποιείται για έλεγχο της ουράς (π.χ. διαγραφή της).

Σήματα (signals)

- Τα σήματα είναι «διακοπές» λογισμικού (software interrupts) οι οποίες διακόπτουν την κανονική λειτουργία μίας διεργασίας.
- Προκαλούνται από διάφορες συνθήκες, π.χ. κάποιο πρόβλημα στο hardware, κάποια παράνομη λειτουργία (π.χ. διαίρεση δια μηδέν ή προσπέλαση σε θέση μνήμης που δεν επιτρέπεται) ή και από τον χρήστη (π.χ. πάτημα CTRL-C).
- Τα περισσότερα σήματα προκαλούν τον άμεσο τερματισμό της διεργασίας, εκτός και αν έχει προβλεφτεί τρόπος «διαχείρισής» τους από την ίδια την εφαρμογή.
- Υπάρχουν πολλά διαφορετικά είδη σημάτων, π.χ.

```
SIGINT 2 /* interrupt */,  SIGQUIT 3 /* quit */,
SIGILL 4 /* illegal instruction */, SIGKILL 9 /* hard kill */,
SIGALRM 14 /* alarm clock */,  SIGCHLD 20 /* to parent on child exit */
```

Διαχείριση σημάτων

- Για να μην τερματίσει η εφαρμογή μας αλλά να «ειδοποιηθεί» για το σήμα και να το διαχειριστεί, πρέπει να έχει ορισθεί ένας **διαχειριστής σήματος (signal handler)**, δηλ. μία συνάρτηση η οποία θα κληθεί αυτόματα αν ποτέ συμβεί το σήμα.
- Ο ορισμός ενός διαχειριστή σήματος γίνεται με απλό (αλλά *ξεπερασμένο* τρόπο) με την κλήση `signal()`.

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

Η οποία βασικά λέει στην `signal()` ότι αν συμβεί το σήμα `sig`, θα πρέπει να κληθεί η συνάρτηση `func`. Αν όλα πάνε καλά η `signal()` επιστρέφει δείκτη στην `func`, αλλιώς `SIG_ERR`.

- Σημείωση: είναι προτιμητέα η χρήση της **`sigaction()`** αντί της `signal()`.

Παράδειγμα διαχειριστή σήματος

- Η παρακάτω εφαρμογή δεν μπορεί να τερματιστεί με CTRL-C διότι έχει ορίσει ένα χειριστή για το σήμα SIGINT:

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void sig_handler(int signo) {
  printf("You cannot stop me with CTRL-C!\n");
}

int main(void) {
  if (signal(SIGINT, sig_handler) == SIG_ERR)
    printf("Can't catch SIGINT\n");
  /* Loop for ever */
  while(1) {
    sleep(1);    /* Let it sleep for a while */
  }
  return 0;
}
```

- Δεν μπορεί να ορισθούν διαχειριστές για όλα τα σήματα.

Τα σήματα ως διαδιεργασιακή επικοινωνία

- Μία διεργασία μπορεί να στείλει σε μία άλλη ένα σήμα.

```
#include <signal.h>
int kill(pid_t pid, int signal);
```

- Υπάρχουν τα `SIGUSR1` και `SIGUSR2`.
- Με αυτό τον τρόπο, αν η διεργασία που το λαμβάνει έχει ορίσει τον αντίστοιχο διαχειριστή, είναι σαν να έχουμε «επικοινωνήσει» στέλνοντάς της έναν (συγκεκριμένο) ακέραιο αριθμό.
- Τα σήματα δεν μπορούν να χρησιμοποιηθούν για ανταλλαγή δεδομένων – το μόνο που μπορεί να «στείλει» ένα σήμα είναι ο εαυτός του και τίποτε άλλο.
- Περισσότερο για συγχρονισμό παρά επικοινωνία.