ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

Εργαστήριο - Άσκηση 4

Διαδιεργασιακή επικοινωνία, Σωληνώσεις στο Unix, Σήματα

Εισαγωγή

Διαδιεργασιακή επικοινωνία (InterProcess Communication, IPC) ονομάζεται ένα σύνολο μηχανισμών που παρέχουν τα λειτουργικά συστήματα, για να διευκολύνουν την ανταλλαγή δεδομένων και τον συγχρονισμό διεργασιών μέσω δομών δεδομένων του πυρήνα. Τέτοιοι μηχανισμοί είναι απαραίτητοι στα σύγχρονα λειτουργικά συστήματα όπου κάθε διεργασία έχει τον δικό της ιδιωτικό χώρο εικονικών διευθύνσεων στον οποίον έχει πρόσβαση μόνο αυτή και ο πυρήνας. Προκειμένου να υπάρχει μία στοιχειώδης προστασία μνήμης μεταξύ διαφορετικών διεργασιών, καμία διεργασία δεν έχει δικαίωμα ανάγνωσης ή εγγραφής στον χώρο διευθύνσεων των υπολοίπων. Όταν δύο διαφορετικές διεργασίες πρέπει να επικοινωνήσουν μεταξύ τους ή να ανταλλάξουν δεδομένα, αυτό γίνεται μέσω του συστήματος αρχείων (π.χ. μία διεργασία να γράψει ένα αρχείο και μία άλλη να το διαβάσει) ή μέσω μίας μεθόδου διαδιεργασιακής επικοινωνίας. Με το προγραμματιστικό μοντέλο των υποδοχών (sockets) οι διεργασίες οι οποίες επικοινωνούν μπορούν να εκτελούνται σε διαφορετικούς υπολογιστές που διασυνδέονται μέσω ενός δικτύου.

Σε πολλές περιπτώσεις ένα εκτελούμενο πρόγραμμα (η μητρική ή γονική διεργασία) δημιουργεί δευτερεύουσες (θυγατρικές) διεργασίες ώστε να εκμεταλλευτεί πιθανά οφέλη από την παράλληλη εκτέλεση τους. Με αυτόν τον τρόπο, σε ένα παράλληλο σύστημα οι υπολογισμοί που απαιτούνται από μία εφαρμογή μπορούν να κατανεμηθούν σε πολλαπλούς επεξεργαστές με τον καθένα να εκτελεί διαφορετική διεργασία, ενώ σε ένα σειριακό σύστημα αν μία διεργασία ανασταλεί (π.χ. σε μία κλήση συστήματος) καθώς περιμένει την απελευθέρωση ενός πόρου (π.χ. πρόσβαση στον σκληρό δίσκο ή μία είσοδο από τον χρήστη), κάποια άλλη διεργασία μπορεί να συνεχίσει τους υπολογισμούς. Είναι φανερό επομένως ότι η διαδιεργασιακή επικοινωνία δεν είναι απαραίτητη μόνο για την ανταλλαγή δεδομένων μεταξύ ανεξάρτητων διεργασιών, αλλά και για τον συντονισμό στενά συνεργαζόμενων διεργασιών οι οποίες εκτελούνται παράλληλα, σε συστήματα πολλαπλών επεξεργαστών, ή ψευδοπαράλληλα, δηλαδή με ταχύτατη και διαφανή εναλλαγή πολλαπλών ταυτοχρόνως εκτελούμενων διεργασιών στον μοναδικό επεξεργαστή.

Η έννοια της διοχέτευσης

Στην πληροφορική σωλήνωση, διοχέτευση ή διασωλήνωση (pipeline) καλείται ένα σύνολο από στοιχεία, υποσυστήματα μιας διαδικασίας επεξεργασίας δεδομένων τα οποία συνδέονται σε σειρά, έτσι ώστε η έξοδος από ένα στοιχείο να αποτελεί είσοδο για το επόμενο στη σειρά. Τα δεδομένα μπορούν επομένως να υφίστανται επεξεργασία παράλληλα από τα διάφορα υποσυστήματα της σωλήνωσης.

Οι βασικές σωληνώσεις σχετικές με υπολογιστές είναι :

- η σωλήνωση εντολών όπως η κλασική σωλήνωση RISC, που χρησιμοποιείται στην Κεντρική Μονάδα Επεξεργασίας επιτρέποντας να εκτελούνται παράλληλα τα στάδια διαφορετικών εντολών αυξάνοντας την απόδοση του υπολογιστή. Στη σωλήνωση αυτή τα στοιχεία της επεξεργασίας είναι μέρη του επεξεργαστή που υλοποιούν τα επιμέρους στάδια εκτέλεσης μια εντολής (προσαγωγή, αποκωδικοποίηση, υπολογισμός τελεστών, προσαγωγή τελεστών, εκτέλεση εντολής κτλ.). Πρόκειται για μία μέθοδο παραλληλισμού εντολών στο υλικό του επεξεργαστή.

- η σωλήνωση γραφικών η οποία συναντάται στις περισσότερες κάρτες γραφικών, οι οποίες αποτελούνται από πολλαπλές μονάδες αριθμητικής ή πλήρεις επεξεργαστές που υλοποιούν τα διαφορετικά στάδια λειτουργιών γραφικής απόδοσης(rendering) όπως προοπτική προβολή, ψαλίδισμα παραθύρων, υπολογισμούς φωτισμού κτλ.
- η σωλήνωση λογισμικού η οποία αποτελείται από διεργασίες κατάλληλα διαμορφωμένες ώστε το ρεύμα δεδομένων εξόδου της μιας να είναι ρεύμα δεδομένων εξόδου για την επόμενη.
- η σωλήνωση (pipe) ως ένας μηχανισμός διαδιεργασιακής επικοινωνίας. Έτσι υλοποιείται εσωτερικά η σωλήνωση λογισμικού στα λειτουργικά συστήματα Unix.

Σωλήνωση στο UNIX

Η παλαιότερη μέθοδος διαδιεργασιακής επικοινωνίας στο Unix είναι οι σωληνώσεις (pipes). Πρόκειται για δομές δεδομένων του πυρήνα που επιτρέπουν σε δύο συγγενείς διεργασίες να ανταλλάσουν αμφίδρομα δεδομένα. Εμφανίζονται στον χώρο του χρήστη ως ζεύγη περιγραφέων αρχείων, χωρίς να αντιστοιχούν σε πραγματικά αρχεία, όπου ο ένας περιγραφέας είναι άκρο εγγραφής και ο άλλος άκρο ανάγνωσης. Οι συνήθεις κλήσεις συστήματος για Είσοδο /΄Εξοδο σε αρχεία (read(), write(), close()) βρίσκουν εφαρμογή και εδώ. Σε κάθε σωλήνωση μπορούν να αντιστοιχούν πολλαπλά ζεύγη περιγραφέων σε διαφορετικές διεργασίες, υλοποιώντας έτσι τη διαδιεργασιακή επικοινωνία. Αυτή η πολλαπλότητα γίνεται εφικτή μέσω της κατασκευής νέων διεργασιών (μοντέλο fork(), όπου η θυγατρική κληρονομεί όλους τους ανοικτούς περιγραφείς της γονικής διεργασίας). Είναι ευθύνη του προγραμματιστή να κλείσει τα άκρα ανάγνωσης ή εγγραφής στις αντίστοιχες διεργασίες ώστε να επιτύχει την επιθυμητή συμπεριφορά (π.χ. μία διεργασία να γράφει και η γονική της να διαβάζει τη σωλήνωση). Το μειονέκτημα είναι ότι με σωληνώσεις μόνο συγγενείς διεργασίες μπορούν να επικοινωνήσουν (γονική με θυγατρική ή αδελφές διεργασίες μεταξύ τους).

Η έννοια των σωληνώσεων μας είναι γνωστή από την εμπειρία μας με το φλοιό. Για παράδειγμα, η εντολή:

```
ls | wc
```

δημιουργεί μια σωλήνωση ανάμεσα στις διεργασίες ls και wc. Οι δυο διεργασίες ξεκινούν ταυτόχρονα και ο φλοιός συνδέει την standard έξοδο (stdout) του προγράμματος ls με τη standard είσοδο (stdin) του προγράμματος wc. Η σύνδεση επιτυγχάνεται δημιουργώντας μια σωλήνωση ανάμεσα στις δυο διεργασίες.

Μια σωλήνωση είναι ένα κανάλι εισόδου/εξόδου μιας κατεύθυνσης. Τα δεδομένα που γράφονται, από μια ή περισσότερες διεργασίες, στο ένα άκρο (είσοδο) της σωλήνωσης αποθηκεύονται σε κάποιο απομονωτή και μπορούν να διαβαστούν από άλλες διεργασίες στην έξοδο της σωλήνωσης. Τα δεδομένα διαβάζονται πάντα με τη σειρά που γράφονται, είναι δηλαδή η σωλήνωση μια δομή FIFO.

Για να δημιουργήσουμε μια σωλήνωση εκτελούμε την κλήση:

```
#include <unistd.h>
int pipe(int pd[2]);
```

Η κλήση `pipe()` επιστρέφει δυο περιγραφητές αρχείων στα στοιχεία `pd[0]` και `pd[1]` του πίνακα `pd`. Ο περιγραφητής `pd[1]` χρησιμοποιείται για εγγραφή δεδομένων στο ένα άκρο της σωλήνωσης, ενώ ο pd[0] χρησιμοποιείται για ανάγνωση δεδομένων από το άλλο άκρο της. Συνήθως, μετά τη δημιουργία μιας σωλήνωσης με τη κλήση pipe(), η διεργασία-γονέας δημιουργεί τον κατάλληλο αριθμό παιδιών με τη βοήθεια κλήσεων fork(). Όπως είναι γνωστό, οι διεργασίες- παιδιά κληρονομούν όλους τους περιγραφητές αρχείων του πατέρα, άρα και τις σωληνώσεις. Με τη βοήθεια κλήσεων write(pd[1],…) οι διεργασίες-παιδιά μπορούν να γράφουν δεδομένα στη σωλήνωση, σαν να έγραφαν σε κάποιο κοινό αρχείο. Αντίστοιχα με κλήσεις read(pd[0],…) δεδομένα μπορούν να διαβαστούν από το άλλο άκρο της σωλήνωσης.

Το μέγεθος του απομονωτή που χρησιμοποιείται στις σωληνώσεις είναι συνήθως 4096 bytes (σε άλλα συστήματα 5120 bytes). Εκτέλεση κλήσης write() σε μια σωλήνωση ενώ ο απομονωτής της δεν διαθέτει αρκετό ελεύθερο χώρο, προκαλεί μπλοκάρισμα της διεργασίας μέχρις ότου δημιουργηθεί αρκετός χώρος. Η κλήση read(), ως γνωστό, μπλοκάρει πάντα εφ' όσον δεν υπάρχουν διαθέσιμα δεδομένα στον απομονωτή (άδεια σωλήνωση). Κλήσεις read() σε άδειες σωληνώσεις στις οποίες όλοι οι περιγραφητές εγγραφής είναι κλειστοί (δηλ. όλες οι διεργασίες που χρησιμοποιούν τη σωλήνωση εκτέλεσαν close(pd[1])), επιστρέφουν End-of-file (EOF). Κλήσεις write() σε σωληνώσεις στις οποίες όλοι οι περιγραφητές ανάγνωσης είναι κλειστοί προκαλούν την αποστολή του σήματος SIGPIPE στην αντίστοιχη διεργασία.

Παράδειγμα: Στο επόμενο παράδειγμα, το πρόγραμμα main δημιουργεί μια σωλήνωση pd και στη συνέχεια δημιουργεί N διεργασίες-παιδιά. Οι διεργασίες- παιδιά (παραγωγοί) στέλνουν η καθεμιά από ένα μήνυμα στον γονέα και στη συνέχεια τερματίζουν την εκτέλεσή τους. Ο γονέας (καταναλωτής) διαβάζει όλα τα μηνύματα από το άλλο άκρο της σωλήνωσης, τα τυπώνει στη standard έξοδο με τη σειρά που τα διαβάζει και τερματίζει επίσης την εκτέλεσή του. Παρατηρείστε ότι όλες οι διεργασίες-παιδιά κλείνουν το άκρο ανάγνωσης της σωλήνωσης pd[0] ενώ ο γονέας κλείνει το άκρο εγγραφής pd[1]. Έτσι δημιουργείται ένα κανάλι μιας κατεύθυνσης από τα παιδιά προς το γονέα.

```
#define BUFSIZ 100
int N=10;
int pd[2];

int main(){
  int i, n;
  char buf[BUFSIZ];
  if ( pipe(pd) < 0 ) error("'cant open pipe");

  for (i=1; i<=N; i++)
    if ( fork() == 0 ) child(i); /* Δημιουργίαπαιδιών  */

  close(pd[1]);                 /* Κλείσιμοάκρουεγγραφής   */

  printf("FATHER: I νreceied from children:\n\n");
  while ( (n=read(pd[0], buf, BUFSIZ)) > 0 )
    write(1, buf, n);           /* 1=stdout */
  close(pd[0]);
}

int child(int i){
  char buf[BUFSIZ];

  close(pd[0]);                 /* Κλείσιμοάκρουανάγνωσης   */

  sprintf(buf, "Hello from child %2d (pid=%5d)\n", i, getpid());
  write(pd[1], buf, strlen(buf)+1);
  close(pd[1]);
  exit(0);
}
```

Συνοψίζουμε τα βήματα εγκαθίδρυσης επικοινωνίας διεργασιών μέσω σωληνώσεων:

1. Η διεργασία-γονέας εκτελεί την κλήση `pipe()`.
2. Η διεργασία-γονέας δημιουργεί διεργασίες-παιδιά μέσω κλήσεων fork().

3. Οι διεργασίες-παραγωγοί κλείνουν το άκρο ανάγνωσης της σωλήνωσης.
4. Οι διεργασίες-καταναλωτές κλείνουν το άκρο εγγραφής της σωλήνωσης.
5. Μεταφέρονται δεδομένα μέσω κλήσεων `read()` και `write()`.
6. Κάθε διεργασία κλείνει τα άκρα της σωλήνωσης που παρέμειναν ανοικτά.

Ακολουθεί ένα παράδειγμα χειρισμού σωληνώσεων σε γλώσσα προγραμματισμού C:

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#define MAXLINE 512
int main(void)
{
  int n, fd[2];
  pid_t pid;
  char line[MAXLINE];
  if ( pipe(fd) < 0)
    { perror("creating pipe"); exit(1); } /* Αποτυχίακατασκευήςτηςσωλήνωσης    */
  if ( (pid = fork()) < 0)
    { perror("cannot fork"); exit(1); } /* Αποτυχίακατασκευήςτηςθυγατρικήςδιεργασίας
                                          */
  else if (pid > 0) /* γονικήδιεργασία  */
    {
      /* κλείσιμοτουάκρουανάγνωσης    */
      close(fd[0]);

      /* κλήσηεγγραφήςσεαρχείο    - εδώχρησιμοποιείταιγιαεγγραφήστοανοιχτόάκροτηςσωλήνωσης
             */
      write(fd[1], "message through pipe\n", 21);
      close(fd[1]);
    }
  else /* θυγατρικήδιεργασία  */
    {
      /* κλείσιμοτουάκρουεγγραφής    */
      close(fd[1]);
      /* κλήσηανάγνωσηςαπόαρχείο    - εδώχρησιμοποιείταιγιαανάγνωσηαπότοανοιχτόάκροτηςσωλήνωσης
              */
      n = read(fd[0], line, MAXLINE);
      write(STDOUT_FILENO, line, n); close(fd[0]);
    }
  exit(0);
}
```

Οποιοσδήποτε αριθμός διεργασιών μπορεί να χρησιμοποιεί μια σωλήνωση για γράψιμο, ανάγνωση ή γράψιμο και ανάγνωση ταυτόχρονα. Οι διεργασίες αυτές ωστόσο πρέπει να είναι όλες κοινοί απόγονοι της διεργασίας στην οποία εκτελέστηκε η κλήση `pipe()`. Δυστυχώς οι πρώτες εκδόσεις του UNIX δεν διέθεταν standard δυνατότητα επικοινωνίας ανάμεσα σε άσχετες μεταξύ τους διεργασίες (π.χ. διεργασίες διαφορετικών χρηστών). Τέτοιες δυνατότητες εμφανίστηκαν στην έκδοση 4.2 BSD μέσα από τον μηχανισμό των sockets και σήμερα παρέχονται πλέον από όλες τις εκδόσεις του UNIX.

Σήματα

Οι διεργασίες στο UNIX δεν αποτελούν "κλειστά συστήματα", ξεκομμένα και ανεπηρέαστα από τη εξέλιξη των υπολοίπων διεργασιών του συστήματος. Αντίθετα κάθε διεργασία επηρεάζει και επηρεάζεται από τις υπόλοιπες διεργασίες με διάφορους τρόπους. Μια διεργασία λ.χ. που έχει ζητήσει να διαβάσει δεδομένα από το δίσκο, αφού πρώτα περάσει τις κατάλληλες παραμέτρους στο πρόγραμμα-οδηγό του δίσκου (disk driver), αναστέλλει την λειτουργία της περιμένοντας να γίνουν διαθέσιμα τα δεδομένα. Μόλις το πρόγραμμα-οδηγός του δίσκου μεταφέρει τα δεδομένα από το δίσκο στη μνήμη θα στείλει ένα σήμα (signal) στην αρχική διεργασία το οποίο θα τη θέσει πάλι σε ενέργεια, ειδοποιώντας την ότι τα δεδομένα είναι διαθέσιμα.

Όταν μια διεργασία δημιουργεί διεργασίες-παιδιά και στη συνέχεια εκτελέσει την κλήση:

```
wait(&status);
```

τότε η διεργασία-γονέας αναστέλλει τη λειτουργία της μέχρις ότου κάποιο από τα παιδιά της τερματίσει. Στην περίπτωση αυτή, το ΛΣ στέλνει ένα σήμα (SIGCHLD) στη διεργασία-γονέα, το οποίο την ξυπνάει από το λήθαργο ώστε να συνεχίσει την εκτέλεσή της.

Τέλος, όταν επιθυμούμε να διακόψουμε ή να τερματίσουμε πρόωρα ένα πρόγραμμα που εκτελούμε από τον φλοιό, πατούμε τα πλήκτρα <CTRLZ> ή <CTRL>C αντίστοιχα. Αυτό έχει σαν αποτέλεσμα να στείλει ο φλοιός στις αντίστοιχες διεργασίες τα σήματα SIGTSTP ή SIGTERM αντίστοιχα, τα οποία τις αναγκάζουν να διακόψουν ή να τερματίσουν τη λειτουργία τους.

Από τα παραπάνω παραδείγματα έγινε φανερό ότι στο UNIX μια μέθοδος αλληλεπίδρασης ανάμεσα σε διεργασίες είναι με τη βοήθεια του μηχανισμού των σημάτων. Τα σήματα είναι, όπως θα δούμε, κάτι σαν ένα σήμα διακοπής προερχόμενο από άλλες διεργασίες και δεν επαρκούν για πραγματική επικοινωνία διεργασιών.

Στο αρχείο <signal.h> βρίσκονται οι ορισμοί των διαφόρων ειδών σημάτων. Τα σήματα αναπαρίστανται με τη βοήθεια μικρών ακεραίων. Στην πράξη ωστόσο χρησιμοποιούμε πάντα τις αντίστοιχες συμβολικές ονομασίες τους. Τα πιο χρήσιμα σήματα δίνονται στον ακόλουθο πίνακα:

Κωδικός	Συμβολισμός	Κατηγορία	Πότε στέλνεται / Σημασία
1	SIGHUP	Exit	Διακοπή της γραμμής επικοινωνίας με το τερματικό
2	SIGINT	Exit	Διακοπή του προγράμματος (πλήκτρο DEL)
3	SIGQUIT	Core	Πρόωρος τερματισμός διεργασίας (πλήκτρα <CTRL>\)
4	SIGILL	Core	Παράνομη εντολή προγράμματος (illegal instruction)
5	SIGTRAP	Core	Παγίδευση εκτέλεσης για έλεγχο του προγράμματος
6	SIGABRT	Core	Διακοπή εκτέλεσης του προγράμματος
7	SIGEMT	Core	Παγίδευση εξομοίωσης
8	SIGFPE	Core	Λάθος σε πράξη κινητής υποδιαστολής
9	SIGKILL	Exit	Άμεσος τερματισμός διεργασίας
10	SIGBUS	Core	Γενικό σφάλμα προγράμματος
11	SIGSEGV	Core	Παράνομη αναφορά σε τμήμα μνήμης
12	SIGSYS	Core	Αναφορά σε ανύπαρκτη πρωτογενή κλήση
13	SIGPIPE	Exit	Προσπάθεια εγγραφής σε κλεισμένη σωλήνωση

Κωδικός	Συμβολισμός	Κατηγορία	Πότε στέλνεται / Σημασία
14	SIGALRM	Exit	Λήξη κάποιου προκαθορισμένου χρονικού διαστήματος
15	SIGTERM	Exit	Τερματισμός διεργασίας (πλήκτρα `<CTRL>C`)
16	SIGUSR1	Exit	Σήμα (αριθμός 1) οριζόμενο από το χρήστη
17	SIGUSR2	Exit	Σήμα (αριθμός 2) οριζόμενο από το χρήστη
18	SIGCHLD	Ignore	Τερματισμός ή διακοπή κάποιας διεργασίας-παιδί
19	SIGPWR	Ignore	Κατάσταση επανεκκίνησης συστήματος
20	SIGWINCH	Ignore	Αλλαγή μεγέθους παραθύρου
21	SIGURG	Ignore	Επείγουσα κατάσταση για κάποιο socket
22	SIGIO	Exit	Επιτρέπεται Ε/Ε σε κάποια περιφερειακή μονάδα
23	SIGSTOP	Stop	Άμεση αναστολή εκτέλεσης διεργασίας με σήμα
24	SIGTSTP	Stop	Αναστολή εκτέλεσης διεργασίας από χρήστη
25	SIGCONT	Ignore	Συνέχιση διεργασίας που έχει διακοπεί

Αν σε ένα πρόγραμμα C επιθυμούμε να κάνουμε χρήση των πρωτογενών κλήσεων που θα παρουσιάσουμε σ'αυτή την παράγραφο, τοποθετούμε στην κορυφή του πηγαίου αρχείου μας την εντολή:

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

Προκειμένου να στείλουμε ένα σήμα σε κάποια διεργασία χρησιμοποιούμε την κλήση kill():

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

όπου pid είναι ο αριθμός ταυτότητας της επιθυμητής διεργασίας και `sig` είναι ο κωδικός ή η συμβολική ονομασία του σήματος που θέλουμε να στείλουμε. Σημειώνουμε ότι το όνομα της κλήσης `kill()` είναι λίγο παραπλανητικό καθώς η αποστολή ενός σήματος σε μια διεργασία δε σημαίνει αυτόματα και τον τερματισμό της διεργασίας αυτής. Όπως είναι λογικό, η κλήση kill() επιτρέπεται να στείλει σήματα μόνον σε άλλες διεργασίες του ιδίου χρήστη. Μια διεργασία επιτρέπεται να στείλει σήμα και στον εαυτό της.

Παράδειγμα: Η κλήση

```
kill(pid, SIGSTOP);
```

διακόπτει τη διεργασία pid.

Η "συμπληρωματική" κλήση της `kill()` είναι η κλήση:

```
#include <unistd.h>
int pause(void);
```

Με την κλήση `pause()` μια διεργασία αναστέλλει τη λειτουργία της μέχρις ότου λάβει κάποιο σήμα.

Κάθε διεργασία που περιμένει να λάβει σήματα, έχει τη δυνατότητα να καθορίσει κάποια συνάρτηση της C η οποία θα εκτελεστεί μόλις ληφθεί το σήμα. Με την παραλαβή κάποιου σήματος, το πρόγραμμα εγκαταλείπει την κανονική ροή του και εκτελεί την καθορισμένη συνάρτηση χειρισμού του σήματος.

Κατά τη διάρκεια της εκτέλεσης της συνάρτησης αυτής, "μπλοκάρεται" η παραλαβή δεύτερου σήματος όμοιου με το αρχικό. Αν η συνάρτηση χειρισμού επιστρέφει κανονικά, τότε η εκτέλεση του κυρίως προγράμματος συνεχίζεται από το σημείο που διακόπηκε. Η όλη διαδικασία θυμίζει έντονα τη διαδικασία εξυπηρέτησης διακοπών με τη διαφορά ότι:

- τα σήματα προέρχονται συνήθως από άλλες διεργασίες και όχι από εξωτερικές συσκευές και
- ο χειρισμός τους γίνεται σε επίπεδο γλώσσας υψηλού επιπέδου. Παράδειγμα: Με την κλήση:

```
int sigtype;
int (*previous)(), handler();
previous = signal(sigtype, handler);
```

καθορίζουμε ώστε η συνάρτηση handler να εκτελεστεί μόλις ληφθεί το σήμα sigtype από την καλούσα διεργασία. Η τιμή previous που επιστρέφει η κλήση signal() είναι ένας δείκτης προς την προηγούμενα καθορισμένη συνάρτηση χειρισμού του σήματος αυτού (χρήσιμη για πιθανό μετέπειτα restore).

Στο αρχείο <signal.h> υπάρχουν τα συμβολικά ονόματα μερικών προκαθορισμένων συναρτήσεων χειρισμού σημάτων. Αυτά είναι:

SIG_IGN Αγνοεί το αντίστοιχο σήμα (ignore)

SIG_DFL Εκτελεί μια προκαθορισμένη (default) ενέργεια για κάθε σήμα. Στον πίνακα των σημάτων που δώσαμε παραπάνω κατατάξαμε τα σήματα σε 4 κατηγορίες. Οι κατηγορίες αυτές καθορίζουν ποιες είναι οι προκαθορισμένες ενέργειες για κάθε σήμα. Αναλυτικά:

ΚΑΤΗΓΟΡΙΑ Exit: Η διεργασία τερματίζεται

ΚΑΤΗΓΟΡΙΑ Core: Η διεργασία τερματίζεται και δημιουργείται ένα αρχείο core (εικόνα του χώρου διευθύνσεων της διεργασίας τη στιγμή του τερματισμού) στον τρέχοντα κατάλογό μας. Το αρχείο core χρησιμοποιείται από τον debugger adb του UNIX.

ΚΑΤΗΓΟΡΙΑ Stop: Η διεργασία διακόπτεται

ΚΑΤΗΓΟΡΙΑ Ignore: Το σήμα αγνοείται

Παραδείγματα: Η κλήση:

```
signal(SIGTERM, SIG_IGN);
```

αγνοεί το σήμα SIGTERM, δηλαδή καθιστά αδύνατο τον τερματισμό της διεργασίας μας με <CTRL>C. Οι εντολές:

```
int (* oldterm)(), onintr();
    :
oldterm = signal(SIGTERM, onintr);
    :
onintr() {
  unlink(tempfile);
  exit(1);
}
```

καθορίζουν μια συνάρτηση onintr χειρισμού του σήματος SIGTERM. Στην περίπτωση αυτή, το πάτημα των πλήκτρων <CTRL>C προκαλεί πράγματι τον τερματισμό της διεργασίας, καλείται ωστόσο η συνάρτηση onintr για να "καθαρίσει" πιθανές εκκρεμότητες της διεργασίας (π.χ. να σβήσει το προσωρινό αρχείο tempfile με την κλήση unlink()) προτού τερματίσει με την κλήση exit().

Τα σήματα SIGKILL και SIGSTOP προκαλούν τον άμεσο τερματισμό ή διακοπή αντίστοιχα των διεργασιών στις οποίες στέλνονται. Δεν είναι δυνατό να ορίσουμε συναρτήσεις χειρισμού των σημάτων αυτών, ούτε και να τα αγνοήσουμε. Τα σήματα αυτά χρησιμοποιούνται σε έκτακτες περιπτώσεις, όταν δεν είναι δυνατή η διακοπή μιας διεργασίας με ηπιότερα μέσα. Ένα οποιοδήποτε σήμα μπορεί να σταλεί σε μια διεργασία από τον φλοιό με την εντολή:

```
kill –κωδικόςσήματος_ αριθμόςδιεργασίας _
```

Αναφέραμε παραπάνω ότι μια διεργασία μπορεί να στείλει σήματα και στον εαυτό της. Αυτό βρίσκει συχνά χρήσιμες εφαρμογές. Για παράδειγμα η κλήση:

```
#include <unistd.h>
unsigned alarm(unsigned sec);
```

στέλνει το σήμα SIGALRM στη διεργασία που την καλεί sec δευτερόλεπτα μετά την κλήση της. Στο μεταξύ η διεργασία είναι ελεύθερη να συνεχίσει την εκτέλεσή της. Πρόκειται για μια πολύ χρήσιμη δυνατότητα με την οποία μπορούμε να πετύχουμε ενεργοποίηση μιας διεργασίας σε περιοδικά διαστήματα, καθορισμό μεγίστου επιτρεπτού (πραγματικού) χρόνου εκτέλεσης διεργασίας, κλπ.

Η ανεπάρκεια των σημάτων για το συγχρονισμό διεργασιών

Ο μηχανισμός των σημάτων του UNIX θυμίζει επιφανειακά τους σηματοφορείς του Dijkstra. Πραγματικά, θα μπορούσε να αντιστοιχίσει κανείς την κλήση pause() στη λειτουργία P και την κλήση kill() στη λειτουργία V ενός δυαδικού σηματοφορέα. Ωστόσο, η υλοποίηση και η λειτουργία των σημάτων παρουσιάζει θεμελιώδεις διαφορές από εκείνη των σηματοφορέων και, όπως θα διαπιστώσουμε, παρουσιάζεται αναξιόπιστη και ανεπαρκής για τον αποτελεσματικό συγχρονισμό διεργασιών.

Κατ' αρχήν σημειώνουμε ότι η αποστολή ενός σήματος, αφυπνίζει ταυτόχρονα όλες τις διεργασίες οι οποίες το περιμένουν. Η μετέπειτα σειρά εκτέλεσης των αφυπνισμένων διεργασιών εξαρτάται από το χρονοδρομολογητή και είναι απροσδιόριστη. Προφανώς μια τέτοια κατάσταση είναι απαράδεκτη αν επιθυμούσαμε π.χ. να χρησιμοποιήσουμε τα σήματα για να πετύχουμε αμοιβαίο αποκλεισμό.

Ακόμη πιο σημαντικό ωστόσο είναι το γεγονός ότι τα σήματα του UNIX δεν έχουν συσχετισμένη μαζί τους κάποια λέξη της μνήμης, η οποία να αλλάζει περιεχόμενα όταν συμβαίνει κάποιο γεγονός που προκαλεί την αποστολή τους. Αντίθετα, τα σήματα στέλνονται αμέσως σε όλες τις καθορισμένες διεργασίες. Αν κάποιες από αυτές τα "περιμένουν" και έχουν καθορίσει συναρτήσεις χειρισμού τους, έχει καλώς. Αν πάλι όλες οι διεργασίες τα αγνοήσουν, τότε το σήμα είναι σαν να μη στάλθηκε ποτέ. Κανένα ίχνος του δεν παραμένει στο σύστημα.

Η υλοποίηση αυτή εγκυμονεί ένα σοβαρό κίνδυνο. Υπάρχει περίπτωση μια διεργασία να αποφασίσει να "περιμένει" για κάποιο σήμα, το οποίο να της σταλεί προτού αυτή εκτελέσει τις κατάλληλες κλήσεις signal() ή pause(). Στην περίπτωση αυτή, η διεργασία θα περιμένει για κάποιο σήμα που ήδη στάλθηκε και που μπορεί να μην της ξανασταλεί ποτέ.

Εξαίρεση στον κανόνα της αναξιοπιστίας αποτελεί η κλήση wait, η οποία όπως αναφέραμε επιστρέφει -1 αν μια διεργασία-παιδί έχει ήδη τερματιστεί, με άλλα λόγια αν έχει ήδη σταλεί το αντίστοιχο σήμα SIGCHLD. Η ιδιότητα αυτή της wait() μπορεί να χρησιμοποιηθεί για να αποφύγουμε τον κίνδυνο που αναφέραμε στην προηγούμενη παράγραφο. Η γενική μέθοδος αναπτύσσεται μέσα από ένα παράδειγμα: Υποθέτουμε ότι σε ένα τμήμα της, η εφαρμογή μας στέλνει ένα σήμα (αίτηση) σε μια άλλη διεργασία και στη συνέχεια εκτελεί την κλήση pause() περιμένοντας να λάβει απάντηση. Η απλή υλοποίηση:

```
:
kill(pid, τύποςσήματοςαίτησης__);
```

```
pause();
:
```

δεν είναι σίγουρη γιατί η απάντηση μπορεί να φθάσει πριν εκτελεστεί η κλήση pause(). Ο αναγνώστης καλείται να εξηγήσει γιατί η ακόλουθη υλοποίηση είναι απόλυτα σίγουρη, με την προϋπόθεση ότι η απάντηση στέλνεται πάντα μετά την λήψη της αιτήσεως.

```
:
previous = signalτύπος(σήματοςαπάντησης__, onintr);
if ( fork() == 0 )
  pause();
else {
  signalτύπος(σήματοςαπάντησης__, previous);
  kill(pid, τύποςσήματοςαίτησης__); wait(&status);
}
:
onintr() {
  exit(1);
}
```

Συμπερασματικά παρατηρούμε ότι τα σήματα είναι ανεπαρκή για τον αξιόπιστο μεγάλης κλίμακας συγχρονισμό διεργασιών και οδηγούν σε πολύπλοκα προγράμματα. Για τον πραγματικό συγχρονισμό διεργασιών χρειαζόμαστε σηματοφορείς. Οι συναρτήσεις χειρισμού σηματοφορέων στο UNIX είναι, ωστόσο, εξίσου πολύπλοκες καθώς κάθε ενέργεια γίνεται σε πίνακες (ουσιαστικά ομάδες) σηματοφορέων, ενώ και οι παρεχόμενες ενέργειες είναι αρκετές και κωδικοποιημένες. Η απλούστερη αντιμετώπιση είναι η εξομοίωση των σηματοφορέων με τη βοήθεια των παρεχομένων δυνατοτήτων του UNIX όπως ακριβώς περιγράφεται παρακάτω.

Ασκηση – Δραστηριότητα

Στην παράγραφο αυτή παρουσιάζουμε το πρόγραμμα `timeout` στο οποίο χρησιμοποιούνται όλες σχεδόν οι κλήσεις δημιουργίας και τερματισμού διεργασιών και σημάτων. Το πρόγραμμα timeout δέχεται σαν παραμέτρους ένα χρονικό όριο (σε δευτερόλεπτα) και μια εντολή. Το πρόγραμμα εκτελεί την εντολή σαν διεργασία-παιδί. Αν η εντολή δεν ολοκληρωθεί μέσα στο καθορισμένο χρονικό όριο, το πρόγραμμα την τερματίζει απότομα στέλνοντάς της το σήμα SIGKILL. Ο καθορισμός του χρονικού ορίου γίνεται με την κλήση alarm(). Παράδειγμα χρήσης του προγράμματος είναι το παρακάτω:

```
timeout -3600 myprog file1 &
```

Με την παραπάνω εντολή εκτελείται η εντολή myprog file1 το πολύ για μια ώρα. Ο κώδικας του προγράμματος ακολουθεί:

```
/* timeout: Εκτέλεσηπρογράμματοςμεχρονικόόριο     */
#include <stdio.h>
#include <signal.h>

int pid; /* Αριθμόςταυτότηταςδιεργασίαςπαιδιού  - */

int main( int argc, char *argv[] ) {
  int sec=10, status, onalarm();
  /* Εξέτασετιςπαραμέτρους   */
  if ( argc > 1 && argv[1][0]=='-' ) {sec = atoi(&argv[1][1]);
```

```
    argc--, argv++;
  }i f ( a r g c < 2 )
     error("Usage: %s [-10] command", argv[0]); /* Δημιούργησεμιαδιεργασίαπαιδί  - */
  if ( (pid=fork()) == 0 )
    {/* Εκτέλεσετηνεντολήωςδιεργασίαπαιδί    - */
      execv(argv[1], &argv[1]);
      error("execv %s failed", argv[1]);
    }/* Μόλιςλάβειςτοσήμα    SIGALRM εκτέλεσετησυνάρτηση   onalarm */
  signal(SIGALRM, onalarm);

  /* Στείλετοσήμασε    sec δευτερόλεπτα */
  alarm(sec);

  /* Περίμενετοντερματισμότουπαιδιούομαλό     ( ήανώμαλο )
   * Σεπερίπτωσηανώμαλουτερματισμούτύπωσεκατάλληλομήνυμα       */
  if ( wait(&status)==-1 || (status & 0xFF)!=0 )
    error("%s killed", argv[1]);
}

onalarm() /* Σκότωσετοπαιδίμόλιςλάβειςτοσήμα       SIGALRM */
{
  kill(pid, SIGKILL);
}
```