Πανεπιστήμιο Δυτικής Μακεδονίας
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Λειτουργικά Συστήματα

Ενότητα 7: Χειρισμός PID διεργασιών. Χειρισμός σημάτων. Υλοποίηση χρονομέτρων με σήματα. Ιεραρχία διεργασιών.

Επιβλέπων: Δασυγένης Μηνάς
Παυλίδου Ελένη

Δρ. Μηνάς Δασυγένης
mdasyg@ieee.org
Εργαστήριο Λειτουργικών Συστημάτων
http://arch.ece.uowm.gr/courses/os/

Άδειες Χρήσης

- Το παρόν εκπαιδευτικό υλικό υπόκειται σε άδειες χρήσης Creative Commons.
- Για εκπαιδευτικό υλικό, όπως εικόνες, που υπόκειται σε άλλου τύπου άδειας χρήσης, η άδεια χρήσης αναφέρεται ρητώς.

Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ψηφιακά Μαθήματα στο Πανεπιστήμιο Δυτικής Μακεδονίας**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

PID διεργασιών.

- Διεργασία είναι ένα πρόγραμμα υπό εκτέλεση.
- Κάθε διεργασία χαρακτηρίζεται από έναν ακέραιο, το PID (Process ID – Αριθμός Ταυτότητας Διεργασίας). Δεν αλλάζει ποτέ!
- Κάθε διεργασία έχει μια άλλη διεργασία ως γονέα (parent).
- Οι διεργασίες αποτελούν δέντρο. Ρίζα: διεργασία `init` με `pid=1`.

System calls για διεργασίες (1/2)

- `pid_t fork()`: Δημιουργεί μια νέα διεργασία που είναι αντίγραφο της καλούσας διεργασίας.
- `int execve(prog,argv,envp)`: Αλλάζει το εκτελέσιμο πρόγραμμα της τρέχουσας διεργασίας.
- `pid_t wait(status)`: Ελέγχει και (πιθανώς) περιμένει μέχρι μια διεργασία να τερματίσει.

System calls για διεργασίες (2/2)

- `void _exit(status)`: Τερματίζει την τρέχουσα διεργασία.

- `pid_t getpid()`: Επιστρέφει PID τρέχουσας διεργασίας.

- `pid_t getppid()`: Επιστρέφει το PID γονέα της τρέχουσας διεργασίας

➢ Κλήσεις συστήματος `getpid` και `gtppid`

- `int getpid()`
- `int getppid()`

Επιστρέφουν τις ταυτότητες μιας διεργασίας και του γονέα αντίστοιχα.

getpid()

- Η συνάρτηση `getpid()` επιστρέφει το process id της τρέχουσας διεργασίας, π.χ.

```
#include  <stdio.h>
#include  <unistd.h>   /* For getpid() */
    int main()    {
     printf("Process id: %d\n", getpid());
     return 0;
                  }
```

- Εκτέλεση στο τερματικό:

```
$ ./a.out
Process id: 3757
$ ./a.out
Process id: 3760
```

getppid() (1/2)

- Για να διαβάσουμε τις ταυτότητες από ένα πρόγραμμα C στο UNIX χρησιμοποιούμε τα ακόλουθα:

```
#include pid_t getpid(void); // ProcessID
                                διεργασίας.
pid_t getppid(void); // ProcessID πατέρα.
pid_t getuid(void); // Real UserID
                       διεργασίας.
pid_t getgid(void); // Real GroupID
                       διεργασίας.
```

getppid() (2/2)

```
#include<unistd.h>
   int main()
  {
     printf("My pid = %d. My parent's pid =
%d\n", getpid(), getppid());
     exit(0);
     }
```

My pid = 5456. My parent's pid = 2616
My pid = 2668. My parent's pid = 2616
My pid = 2440. My parent's pid = 2616
! Ο πατέρας είναι το κέλυφος, επομένως δεν αλλάζει τιμή.

Ρουτίνες χειρισμού signals

- `sigaction(int num,newaction,oldaction)`: Θέτει το νέο sigaction για το δεδομένο signal.
- `sigprocmask(how,set,oldset)`: Block και unblock διάφορα signals.
- `sigpending(set)`: Εξετάζει ποια signals (που μπλοκάρονται) είναι σε αναμονή (pending).
- `kill(pid,sig)`: Στέλνει το sig στο process με το pid.
- `sigsuspend(maskset)`: Εμπλέκεται και περιμένει για κάποια συγκεκριμένα signals.

Άλλες περιπτώσεις

➢ Αν ο γονέας τερματίσει πριν το παιδί καλέσει την `exit`:

- Νέος γονέας η `init (pid == 1)`.
- Η `init` θα καλέσει τη `wait`.

➢ Αν το παιδί τερματίσει πριν ο γονέας καλέσει τη `waitpid`:

- Το παιδί είναι στην κατάσταση DEAD.
- Διατηρείται στον πίνακα διεργασιών για να διαβαστεί το `exit status`.
- Η κατάστασή του αλλάζει σε `zombie(Z)`.

kill() (1/4)

- Μια από τις πιο γνωστές συναρτήσεις αποστολής σημάτων είναι η `kill`.

```
int kill(int pid, int sig)
```

  Κλήση συστήματος που στέλνει το σήμα `sig` στη διεργασία `pid`.
  Αν `pid > 0` το σήμα αποστέλλεται στην αντίστοιχη διεργασία.
  Αν το `pid = 0` το σήμα αποστέλλεται σε όλες τις διεργασίες, εκτός από αυτές του συστήματος.

- Η συνάρτηση `kill()` επιστρέφει `0` σε επιτυχή εκτέλεση, `-1` σε περίπτωση σφάλματος .

Kill() (2/4)

- Η `kill()` χρησιμοποιείται από τη συνάρτηση `raise()` ως εξής:

```
kill(getpid(),sig);
```

- **Σημείωση:** Δεδομένου ότι η αποστολή σημάτων ουσιαστικά τερματίζει διεργασίες, είναι λογικό το σύστημα να παρέχει προστασία στις εκτελούμενες διεργασίες αλλιώς ο καθένας θα μπορούσε να τις τερματίσει. Η αποστολή και παραλαβή σημάτων επιτρέπεται μόνο μεταξύ διεργασιών που έχουν τον ίδιο ιδιοκτήτη (χρήστη). Εκτός των σημάτων που αποστέλλει ο διαχειριστής.

Kill() (3/4)

- Παράδειγμα “αυτοκτονίας”:

  Η κλήση:

  ```
  kill(getpid(),SIGINT);
  ```

  στέλνει σήμα τερματισμού στη καλούσα διεργασία, δηλαδή στον εαυτό της. Αυτό αντιστοιχεί στην κλήση της `exit()`.
  Το `ctrl-c` στη γραμμή εντολών ουσιαστικά
  στέλνει `SIGINT` στη διεργασία (εντολή ή εφαρμογή) που εκτελείται.

Kill() (4/4)

- Επίσης υπάρχει και η συνάρτηση:

```
unsigned int alarm(unsigned int seconds)
```

που αποστέλλει το σήμα χρονοδιακοπής (αφύπνισης) SIGALRM στη διεργασία που την καλεί, μετά από δευτερόλεπτα. Αντίστοιχα λειτουργεί στη γραμμή εντολών η εντολή UNIX/Linux sleep.

Signal() (1/3)

- Η παραλαβή σήματος είναι απλή:

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

Η συνάρτηση `signal()` θα καλέσει μια από τις συναρτήσεις `func()` με βάση τη τιμή του σήματος `sig`. Αν η κλήση της συνάρτηση `signal()` είναι επιτυχής τότε επιστρέφει δείκτη σε συνάρτηση `func()`. Αν δεν είναι επιτυχής τότε επιστρέφει `-1` και ενημερώνει.

Signal() (2/3)

- Ο δείκτης σε συνάρτηση `func()` μπορεί να πάρει τρείς τιμές:

- **`SIG_DFL`**-- δείκτη σε προεπιλεγμένη συνάρτηση του συστήματος `SID_DFL()` που τερματίζει τη διεργασία μόλις παραλάβει το σήμα `sig`.
- **`SIG_IGN`--** δείκτη σε προεπιλεγμένη συνάρτηση του συστήματος `SIG_IGN()` που θα αγνοήσει το σήμα `sig` (ΕΚΤΟΣ από το `SIGKILL`).
- **Διεύθυνση συνάρτησης**-- δείκτη σε συνάρτηση που ορίζει ο χρήστης.

Signal() (3/3)

- Οι δείκτες `SIG_DFL` και `SIG_IGN` ορίζονται στη πρότυπη βιβλιοθήκη `signal.h`.

- Έτσι αν θέλουμε να αγνοήσουμε το ctrl-c από το πληκτρολόγιο, γράφουμε:

```
signal(SIGINT, SIG_IGN);
```

- Αν πάλι θέλουμε το `SIGINT` να προκαλεί τερματισμό, γράφουμε:

```
signal(SIGINT, SIG_DFL);
```

Sigprocmask (1/5)

- Για να μπλοκάρουμε ένα σήμα, αρκεί να το βάλουμε στο `signal mask` και για να το ξεμπλοκάρουμε αρκεί να το βγάλουμε από αυτό το σύνολο.

- Το `signal mask` αλλάζει με τη συνάρτηση:
  [επιστρέφει < 0 αν αποτύχει]

```
#include <signal.h>
int sigprocmask(int how, sigset_t *newmask,
sigset_t *oldmask);
```

όπου το `newmask` είναι το νέο σύνολο, ενώ στο `oldmask` θα επιστραφεί το παλαιό (αν `oldmask≠NULL`).

Sigprocmask (2/5)

- Το `how` καθορίζει πώς θα γίνει η αλλαγή.

- `SET_SIGMASK`: το νέο signal mask είναι ακριβώς αυτό που δίνεται στο newmask .

- `SET_SIGBLOCK`: πρόσθεσε στο υπάρχον signal mask ό,τι έχει το newmask.

- `SET_SIGUNBLOCK`: αφαίρεσε από το υπάρχον mask ό,τι έχει το newmask .

Sigprocmask (3/5)

- Το `sigset_t` είναι μια δομή που καταγράφει ποια σήματα μπλοκάρονται.

- Οι συναρτήσεις που μπορούν να προσθαφαιρέσουν σήματα στη δομή αυτή είναι οι εξής:

- `sigemptyset(sigset_t *set)`: Άδειασε το σύνολο. Κανένα σήμα δεν θα μπλοκαριστεί.

- `sigfillset(sigset_t *set)`: Βάλε όλα τα σήματα μέσα στο σύνολο. Όλα μπλοκάρονται.

Sigprocmask (4/5)

- `sigaddset(sigset_t *set, int sigid)`: Πρόσθεσε το σήμα `sigid` στο σύνολο (θα μπλοκάρεται).

- `sigdelset(sigset_t *set, int sigid)`: Αφαίρεσε το σήμα `sigid` από το σύνολο (θα ξεμπλοκάρεται).

- `sigismember(sigset_t *set, int sigid)`: Έλεγξε αν το σήμα `sigid` είναι στο σύνολο.

Sigprocmask (5/5)

- Έλεγχος αν η τρέχουσα signal mask περιέχει το SIGINT και αν ναι, το ξεμπλοκάρουμε:

```
int main() { sigset_t newmask, oldmask; …
  sigprocmask(SIG_SETMASK, NULL, &oldmask);
        /* Just get the current mask */
  if (sigismember(&oldmask,SIGINT)) {
/* Check if SIGINT is blocked */
  sigemptyset(&newmask);
        /* Create an empty set */
   sigaddset(&newmask, SIGINT);
        /* Add the SIGINT signal */
   sigprocmask(SIG_UNBLOCK, &newmask, NULL);
        /* Remove from current mask */ } …
}
```

alarm() (1/3)

- Σήμα `SIGALRM`: Αποστέλλεται σε μία διεργασία από τον πυρήνα όταν εκπνέει ένα χρονόμετρο, το οποίο η ίδια είχε εγκαταστήσει και εκκινήσει νωρίτερα με μία κλήση στη συνάρτηση βιβλιοθήκης `alarm()`, που δέχεται ως όρισμα ακέραιο πλήθος δευτερολέπτων. Διαδοχικές κλήσεις στην `alarm()` επανεκκινούν το χρονόμετρο.

alarm() (2/3)

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

- Η παράμετρος `sec` καθορίζει το χρονικό διάστημα σε δευτερόλεπτα. Όταν ολοκληρωθεί το χρονικό διάστημα προκαλείται σήμα / διακοπή τύπου SIGALRM.

- Αν η συνάρτηση κληθεί πριν τελειώσει το προηγούμενο χρονικό διάστημα που είχε τεθεί, ακυρώνεται το παλιό και ορίζεται νέο διάστημα.

alarm() (3/3)

- Καλώντας την με παράμετρο `0`, ακυρώνεται το χρονόμετρο.
- Επιστρέφει το χρόνο που έμεινε μέχρι να ολοκληρωθεί το προηγούμενο διάστημα.
- Επειδή και η `sleep()` μπορεί να υλοποιηθεί με χρήση των ίδιων χρονομέτρων με την `alarm()`, δεν πρέπει να γίνεται `sleep()` πριν την ολοκλήρωση του χρονικού διαστήματος από την `alarm()`.

fork() (1/4)

- Δυστυχώς οι συναρτήσεις της κατηγορίας exec ΔΕΝ δημιουργούν νέα διεργασία, παρά διατηρούν την τρέχουσα και απλώς τη βάζουν να εκτελέσει ένα άλλο πρόγραμμα.

- Υπάρχει ένας και μοναδικός τρόπος να γίνει αυτό: η κλήση συστήματος `fork()`:

  - Η συνάρτηση αυτή δημιουργεί ένα πανομοιότυπο αντίγραφο της τρέχουσας διεργασίας και οι δύο, πλέον, διεργασίες εκτελούνται ταυτόχρονα και ανεξάρτητα.

fork() (2/4)

- Η διεργασία που κάλεσε την `fork()` ονομάζεται διεργασία-γονέας ενώ η νέα διεργασία ονομάζεται θυγατρική.

- Όταν δημιουργηθεί η διεργασία-παιδί, δεν ξεκινάει να εκτελεί την `main()`. Αρχίζει να εκτελεί, αμέσως μετά το σημείο που έγινε η κλήση στην `fork()`.

  Είναι λες και ο πατέρας και το παιδί να επιστρέφουν μαζί από την `fork`.

fork() (3/4)

- Το παρακάτω πρόγραμμα δείχνει μια απλή χρήση της `fork()`, όπου δύο αντίγραφα του ίδιου προγράμματος εκτελούνται 'ταυτόχρονα' (*πολυεργασία, multitasking*)

```
#include <stdio.h>
#include <unistd.h>
main()
```

Συνέχεια στην επόμενη σελίδα.

fork() (4/4)

```
{
 int some_value;
 printf("Forking process\n");
 fork();
 /* This part of the program is executed by
 two different proceses */
 printf("The process id is %d \n", getpid());
 some_value = getpid() + 10;
 printf("Some value is %d", some_value);
 execl("/bin/ls","/bin/ls","-l",NULL);
/* This line is not executed because of th
execl function */
 printf("This line is not printed\n");
}
```

Τέλος Ενότητας

Πανεπιστήμιο Δυτικής Μακεδονίας