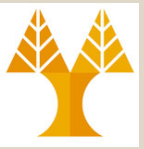
ΕΠΛ421 - Προγραμματισμός Συστημάτων

Διάλεξη 14: Διεργασίες: Έλεγχος & Σήματα (Processes: Control & Signals)

(Κεφάλαιο 8,10 - Stevens & Rago)

Δημήτρης Ζεϊναλιπούρ

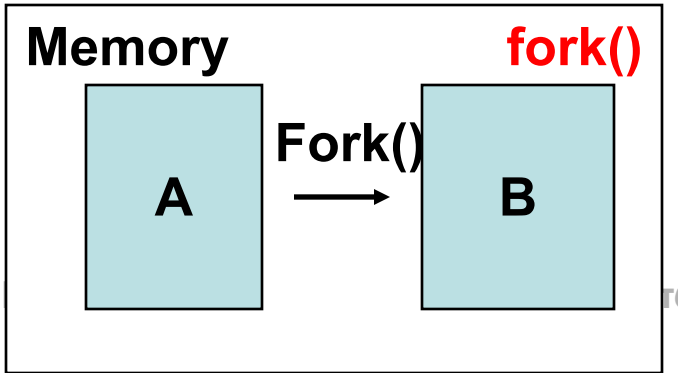
Περιεχόμενο Διάλεξης

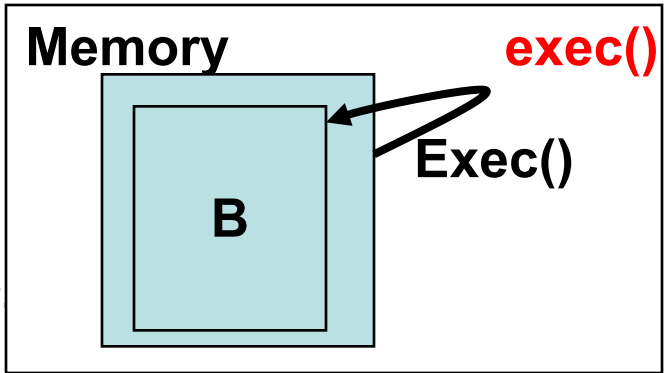


A. Η οικογένεια συναρτήσεων **exec()** (8.10)
B. Η κλήση συστήματος **system()** (8.13)
Γ. **Σήματα στην C** (10.1, 10.2)
Δ. Η κλήση συστήματος **signal()** (10.3)
E. **Σήματα** και **Διεργασίες**
Z. Η Κλήση συστήματος **alarm()**
H. Δια-διεργασιακά Σήματα (**kill()**)

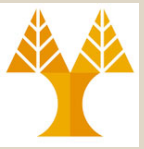
Α. Συναρτήσεις exec()

- Γνωρίζουμε ότι με την κλήση της **fork() κλωνοποίουμε** το process image (στοίβα, κώδικα, σωρό, etc) μιας διεργασίας.
- Οι εντολές exec από την άλλη, **αντικαθιστούν** το process image μιας διεργασίας Α με αυτά ενός εκτελέσιμου αρχείου (π.χ. ls, sort, uniq, οτιδήποτε εκτελέσιμο!)
- Στην συνέχεια η Α θα συνεχίσει να εκτελείται από το "main()" του αρχείου (και όχι από εκεί που έγινε το fork).
- Με την κλήση της exec το Β θα έχει το ίδιο PID, PPID, File Descriptors, alarms, κτλ. του πατέρα Α. (δες σελ. 234)





Α. Παράδειγμα 1 - execl()



Να γραφεί ένα πρόγραμμα το οποίο να παρουσιάζει το περιεχόμενο του καταλόγου “..”, κάνοντας χρήση του execl

Α. Παράδειγμα 1 - execl()

```
/* File: execl_demo.c */
#include <stdio.h> /* For printf */
main()
{
        printf("I am process %d and I will execute an 'ls -l ..'\n", getpid());

        if (execl("/bin/ls", "ls", "-l", "..", NULL) ==  -1) {
                perror("execl");
        }
}
```

Παράδειγμα Εκτέλεσης

```
$ execl_demo
I am process 13974 and I will execute an 'ls -l ..'
total 488
drwx-----x   53 antonis  faculty    155648 Jan  5 09:52 antonis
drwx-----x   47 chryssis faculty      4096 Mar  1 13:46 chryssis
```

Α. Συναρτήσεις exec()

- Υπάρχουν 6 είδη από τα οποία μας ενδιαφέρουν μόνο τα ακόλουθα

```
#include <unistd.h>
int execl(char *path, char *arg0, char *arg1, ... , char *argn, NULL);
int execv(char *path, char *argv[]);
```

arguments

Δηλώνει ότι εδώ τελειώνει το arg list (εναλλακτικός ορισμός (char *)0)

– **Όπου path είναι σχετικό ή απόλυτο μονοπάτι.**

```
int execlp(char *path, char *arg0, char *arg1, ... , char *argn, NULL);
int execvp(char *path, char *argv[]);
```

– **Όπου στο path δεν χρειάζεται να ορίσει την τοποθεσία του εκτελέσιμου εφόσον θα γίνει χρήση της μεταβλητής περιβάλλοντος PATH για να βρεθεί το εκτελέσιμο.** Αυτό είναι χρήσιμο γιατί μπορεί να μην ξέρουμε που ακριβώς βρίσκεται ένα εκτελέσιμο π.χ. sort, uniq, etc).

Και οι τέσσερις επιστρέφουν -1 σε λάθος ενώ σε επιτυχία τίποτα.

- **Η διαφορά της execl(list) / execv(vector)** είναι ότι το execl παίρνει σαν όρισμα: arg0=«όνομα εκτελέσιμου», arg1=«μεταβλητή 1», …. , argn=«μεταβλητή n», arg(n+1)=NULL ενώ η execv argv[0], argv[1], …, argv[n], argv[n+1]=NULL.
- Η ίδια διαφορά ισχύει και για το ζεύγος **execlp(list,PATH) / execvp (vector,PATH).**
- Οι πιο πάνω εντολές είναι **εντολές βιβλιοθήκης** στο Linux που βασίζονται στην **κλήση συστήματος execve** (vector environment).

Α. Παράδειγμα 2 - execvp()

```
#include <stdio.h> /* For printf */
int main()
{
        int pid, status; char *argv[2];
        if ((pid = fork()) == -1) {  /* Check for error */
                perror("fork"); exit(1);
        }
        else if (pid == 0) {
                /* Child process */
                argv[0] = "date"; argv[1] = NULL;
                printf("I am child process %d and I will replace myself by 'date'\n", getpid());
                if (execvp("date", argv) == -1 ) {
                        perror("execvp"); exit(1);
                }
        }
        else {
                /* Parent process */
                if (wait(&status) != pid) {  /* Wait for child
                        perror("wait"); exit(1);
                }
                printf("I am parent process %d\n", getpid());
                printf("Child terminated with exit code %d\n", status >> 8);
        }
        exit(0);
}
```

Περιγραφή

Δημιουργήστε μια διεργασία παιδί η οποία εκτελεί την date. Μόλις ολοκληρώσει το παιδί ο πατέρας συνεχίζει την εκτέλεση του.

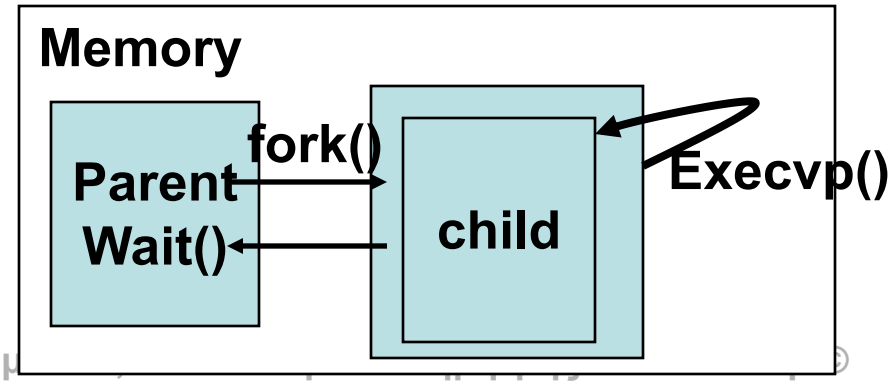
Αποτέλεσμα Εκτέλεσης

```
$./execvp-example2
I am child process 508 and I will replace myself by 'date'
Sun Mar  4 12:50:06 RST 2007
I am parent process 3340
Child terminated with exit code 0
```

Α. Παράδειγμα 3

Υλοποίηση Απλού Κελύφους με execvp

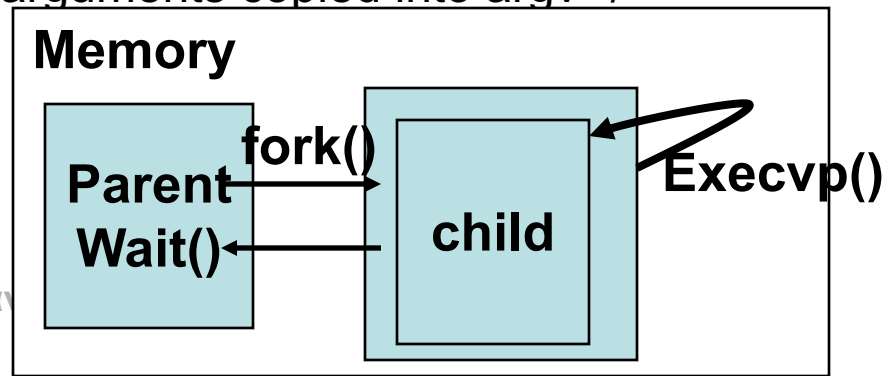
Να γραφεί ένα πρόγραμμα C που να λειτουργεί σαν ένα απλό κέλυφος: Να δέχεται από το πληκτρολόγιο εντολές τις οποίες να αναλαμβάνει να εκτελέσει μια διεργασία-παιδί που θα δημιουργεί.



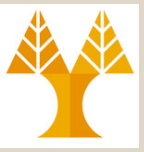
Α. Παράδειγμα 3
Υλοποίηση Απλού Κελύφους

```
/* File: simpleshell.c */
#include <stdio.h> /* For fgets, printf */
void parse(char *buf, char **argv); /* Function prototype */
void execute(char **argv);              /* Function prototype */

int main()
{
        char buf[1024]; char *argv[64];
        while (1) {
            printf("Command: "); /* Get input */
            if (fgets(buf, sizeof(buf), stdin) == NULL) {
                    printf("\n"); exit(0);
            }
            parse(buf, argv); /* Split buf into arguments copied into argv */
            execute(argv);
        }
        return 0;
}
```

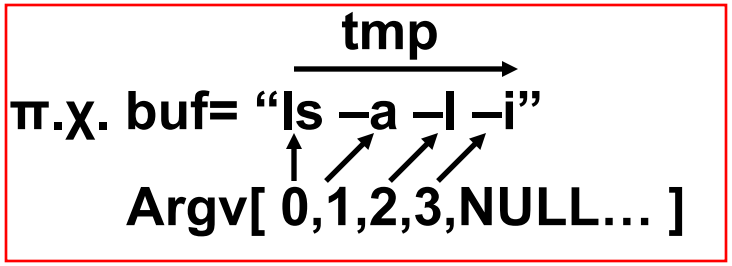
Get user command into buf

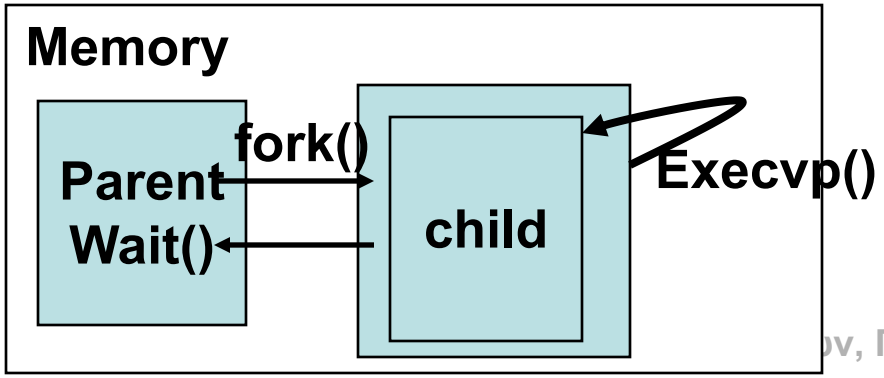


Α. Παράδειγμα 3
Υλοποίηση Απλού Κελύφους



```
// Tokenizes “buf” arguments into “argv” table
void parse(char *buf, char **argv)
{
        char *tmp;  // used to perform a linear scan of buf
        tmp = buf; // initialize buf to the same position with buf
        while (*tmp != '\0')  {
            if ((*tmp == ' ') || (*tmp == '\t') || (*tmp == '\n')) {
                *tmp = '\0';  // replace whitespace with NULL
                *argv++ = buf;  // first set *argv=buf then *argv++
                //printf("%s\n", *--argv);
                buf = ++tmp; // set buf to the next string
            }
            tmp++;
        }
        *argv=NULL;   /*  End of arguments */
}
```





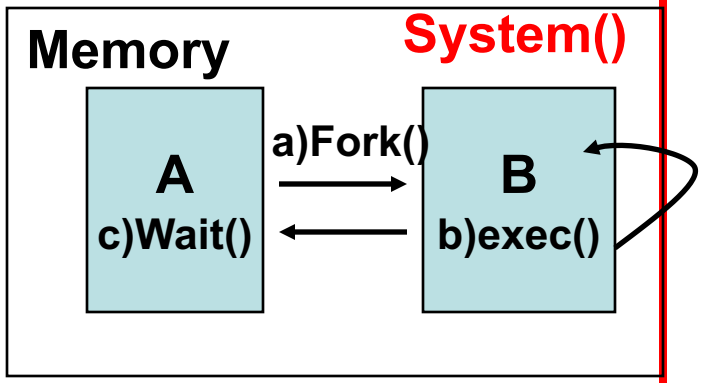

```
void execute(char **argv)
{
    int pid, status;
    if ((pid = fork()) < 0) { /* Create a child */
            perror("fork"); exit(1);
    }
    else if (pid == 0) { /* Child process */
        if (execvp(argv[0], argv) == -1) {
            perror(argv[0]); exit(1);
        }
    } else { /* Parent process */
        if (wait(&status) != pid) {
            perror("wait");  exit(1);
        }
    }
}
```

Vector + PATH

Β. Κλήση Συστήματος system()

- Σε πολλές περιπτώσεις, όπως το προηγούμενο παράδειγμα, θέλουμε το παιδί να εκτελέσει κάποιο υφιστάμενο προγ. π.χ.,

```
#include <unistd.h>
int main() {
     system("ls -al | sort > output.txt");
     exit(0);
}
```




```
int system(const char *cmdstring);
```

επιστρέφει -1 (out of process ή 0 OK)

- Το πρόγραμμα C περιμένει μέχρι να ολοκληρώσει η εντολή system.
- Η εντολή system υλοποιείται ουσιαστικά μέσω **fork (παιδί), exec(παιδί)**, **waitpid (πατέρας περιμένει το παιδί).**

(μελετήστε το παράδειγμα στο σχήμα 8.22 (σελίδα 247)

Β. Παράδειγμα 3 - system()

Πόσες συνολικά διεργασίες *θα δημιουργηθούν από την εκτέλεση του προγράμματος* ***sysexec.c*** *το οποίο χρησιμοποιεί την εντολή* ***system*** *για να καλέσει το πρόγραμμα κελύφους* ***batch.sh****;*

```
$ cat sysexec.c
#include <unistd.h>
int main()
{
   system("./batch.sh");
   exit(0);
}
```

```
$cat ./batch.sh
i=0;
while [ $i -lt 1000 ];
do
   echo -n " $i ";
   ls -al | sort | uniq | rev > /tmp/file;
   ((i++));
done
```

Β. Παράδειγμα 3 - system()

- Η εκτέλεση του *sysexec(7228)* δημιουργεί μια νέα διεργασία *sh(7229) η οποία κάνει 1000 επαναλήψεις.*
- Σε κάθε από τις 1000 επαναλήψεις δημιουργούνται 4 διεργασίες (ls, rev, sort, uniq).
- Θυμίζουμε ότι το ls, rev, sort, uniq δεν είναι built-in εντολές του κελύφους (δηλαδή δεν βρίσκονται μέσα στο εκτελέσιμο αρχείο του shell). Επομένως κάθε κλήση τους δημιουργεί μια νέα διεργασία.
- Επομένως θα έχουμε 4*1000 (ls,sort,rev,uniq) + 1 (*sysexec(7228)*) + 1 (*sh(7229)*) *= 4002 διεργασίες*

Πιο κάτω φαίνονται δυο από τα 1000 στιγμιότυπα με την εντολή pstree.

$ pstree -Gp dzeina

```
sshd(31040)——bash(31042)——sysexec (2709)——sh(7229)—┬—ls(2852)
                                                    ├—rev(2854)
                                                    ├—sort(2853)
                                                    └—uniq(2855)
```

$ pstree -Gp dzeina

```
sshd(31040)——bash(31042)—— sysexec (2709)——sh(7229)—┬—ls(2953)
                                                    ├—rev(2955)
                                                    ├—sort(2954)
                                                    └—uniq(2956)
```

Γ. Διαχείριση Σημάτων (Signals)

- **Signals (software interrupts):** Μικρά μηνύματα, τα οποία στέλνονται σε μια **καθορισμένη** διεργασία ή ομάδες διεργασιών.
- Τα σήματα στέλνονται μεταξύ Διεργασιών (εάν υπάρχουν τα κατάλληλα δικαιώματα, π.χ. kill()) ή από το Πυρήνα στην Διεργασία (π.χ. alarm())
- Όταν λάβει το signal μια διεργασία, **διακόπτει άμεσα την εκτέλεση της**, και διαχειρίζεται το signal.
  - Η διαχείριση του σήματος γίνεται ουσιαστικά με ένα signal handler
  - Εναλλακτικά, η διεργασία μπορεί να αγνοήσει το σήμα (… αλλά όχι όλα τα σήματα)
- Εδώ θα μελετήσουμε τα σήματα στο πλαίσιο της **γλώσσας C** ενώ τα είχαμε δει επίσης στο **πλαίσιο του κελύφους.**

Γ. Διαχείριση Σημάτων (Signals)

- Κάθε σήμα στο UNIX αποτελείται:
  - από ένα **αριθμό (π.χ., 2)** ή αντίστοιχα από ένα ισοδύναμο συμβολικό όνομα (π.χ., SIGINT),
  - Ένα **διαχειριστή σήματος (signal handler)** ο οποίος μπορεί να αντικαταστήσει τη μέθοδο με την οποία ένα πρόγραμμα διαχειρίζεται το σήμα.
- Για να προγραμματίσουμε τα σήματα στην C πρέπει να συμπεριλάβουμε την βιβλιοθήκη

  #include <signal.h>

Δ. Κλήση Συστήματος signal()

```
#include <signal.h>

void cntl_c_handler(int sig)
{
  printf("Just received signal %d\n", sig);
}

main() {
  int I;
  void (*fp)(int); // default sighandler variable
  fp = signal(SIGINT, cntl_c_handler);

  for (i=0;i<5;i++) {
       printf("Still running\n");
       sleep(1);
  }
signal(SIGINT, fp); // επαναφορά default Sighandler
}
```

Αριθμός σήματος που θα ενεργοποιήσει τον handler

Set New signal Handler & Backup previous one in fp

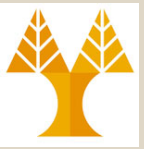
Παράδειγμα Εκτέλεσης

$ ./simplesignal

$ ./a
Still running
Just received signal 2
Still running
Just received signal 2
Still running
Just received signal 2
Still running
Just received signal 2
… εδώ διακόπτεται

* Υποθέστε ότι στέλνουμε το σήμα SIGINT από το κέλυφος

Ε. Σήματα και Διεργασίες



Εάν δηλώσουμε ένα signal handler και κάνουμε fork τότε αυτός ο signal handler ισχύει και για το παιδί.

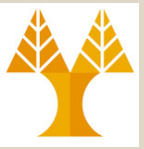
```
#include <signal.h>

void signal_handler(int sig) {
        printf("A signal %d was received by %d \n", sig, getpid());
}

int main(void) {
        pid_t pid;
        /* Install signal handler */
        signal(SIGINT, signal_handler);
        if ( (pid = fork()) < 0)  perror("fork error");
        else if (pid == 0)
                while (1) {
                     printf("child (orphan) still alive\n"); sleep(1);
                }
        else  { printf("parent has finished\n"); }

        exit(0);
}
```

Για να δούμε ότι παραλαμβάνετε από το παιδί μπορούμε να στείλουμε από το κέλυφος $kill -2 4256
Βλέπουμε ότι θα παραληφθεί απευθείας από το παιδί

… … …
child (orphan) still alive
child (orphan) still alive
A signal 2 was received by 4256
child (orphan) still alive
child (orphan) still alive
child (orphan) still alive
… … …

Σημειώστε ότι πρόκειται για ένα ορφανό process

Ε. Σήματα και Διεργασίες
Ασύγχρονη Αποφυγή Zombies με Signals



- Στην διάλεξη 13 (τελευταίο παράδειγμα) είδαμε πως μπορούμε να αποφύγουμε τα zombies με κάποιο σύγχρονο τρόπο.
  - Δηλαδή, εκτελούμε wait() μέχρι να μας επιστρέψει το exit code το παιδί.
  - Αυτό όμως σταματούσε την εκτέλεση του πατέρα (στο wait()) μέχρι να απαντήσει το παιδί. (δηλ. ο πατέρας έκανε blocking wait)
- Τώρα θα δούμε ένα εναλλακτικό, και προγραμματιστικά πιο ορθό τρόπο, για την ασύγχρονη αποφυγή των zombies.
- Θα κάνουμε χρήση του **SIGCHLD**, το οποίο στέλνεται από το παιδί στο πατέρα όποτε το παιδί τερματίσει η σταματήσει προσωρινά την λειτουργία του.

Ε. Σήματα και Διεργασίες
Ασύγχρονη Αποφυγή Zombies με Signals

```
#include <signal.h>
#include <unistd.h>  // STDOUT_FILENO
void signal_handler(int sig) {
   int pid; int status;
   printf("#%d: A signal %d was received \n", getpid(), sig);
   pid = wait(&status);
   printf("#%d: Exit Code %d %d\n", getpid(), pid, status>>8);
}

int main() {
    int pid;
    signal(SIGCHLD, signal_handler); /* Install signal handler */
    pid = fork();
    if (pid == -1) {              /* Check for error */
                perror("fork");  exit(1);
    }
    else if (pid == 0) {       /* The child process */
                printf("#%d: Exit!\n", getpid());
                exit(37);       /* Exit with a silly number */
    }
    else {
                /* The parent process */
                while (1)       /* Never terminate */
                    sleep(1000);
    }
}
```

Τοποθέτηση **wait()** μέσα στον **signal_handler** έτσι ώστε να μπορεί το παιδί να παραδώσει το **exit code** ανεξάρτητα με το πόσο απασχολημένος είναι ο πατέρας. Αυτό περιορίζει τα **zombie processes**.

Αποτέλεσμα Εκτέλεσης

```
$./chldsignal
#5288: Exit!    (Child)
#2920: A signal 20 was received
#2920: Exit Code 5288 37
```

Αυτό προηγούμενα μας δημιουργούσε zombie processes

Τσιν, Δημήτρης Ζεϊναλιπούρ ©

Η. Δια-διεργασιακά Σήματα
Κλήση Συστήματος kill()

- Μέχρι τώρα είδαμε πως μπορούμε να στείλουμε σήματα **από** τον **πυρήνα** (π.χ. SIGALRM) και **από το κέλυφος** (π.χ. SIGINT) σε μια **διεργασία**.
- Τι γίνεται εάν θέλουμε να στείλουμε σήματα μεταξύ διεργασιών;
- **Υπάρχει η κλήση συστήματος kill**

  int kill(int pid, int sigcode);

  Επιστρέφει -1 σε αποτυχία ή 0 σε επιτυχία
  - Η kill είναι μόνο εφικτή εάν η sending διεργασία έχει τον ίδιο ιδιοκτήτη με την receiving διεργασία (ή ο sender είναι root)
  - Για δια-εργασιακή αποστολή σημάτων μπορούμε να χρησιμοποιήσουμε τα μη-δεσμευμένα για άλλο σκοπό SIGUSR1, SIGUSR2.

Η. Δια-διεργασιακά Σήματα
Κλήση Συστήματος kill()

```
#include <stdio.h> /* For printf */
#include <signal.h> /* For SIGTERM, SIGSTOP, SIGCONT */

int main() {
   int pid1, pid2;
   if ((pid1 = fork()) == -1) { perror("fork"); return 1;   }
   else if (pid1 == 0) { // first child loop
     while (1) { /* Infinite loop */
          printf("Process 1 is alive\n"); sleep(1);
     }
   }
   else { // parent
      if ((pid2 = fork()) == -1) {  perror("fork"); return 1;  }
      else if (pid2 == 0) { // second child loop
           while (1) { /* Infinite loop */
            printf("Process 2 is alive\n"); sleep(1);
           }
        }
        // parent's code
        sleep(2); kill(pid1, SIGSTOP); /* Suspend first child */
        sleep(2); kill(pid1, SIGCONT); /* Resume first child */
        sleep(2);
        kill(pid1, SIGTERM); /* Terminate first child */
        kill(pid2, SIGTERM);  /* Terminate second child */
   }
   return 0;
}
```

Περιγραφή

Δημιουργούμε δυο παιδιά τα οποία εκτελούν από ένα άπειρο βρόχο. Στη συνέχεια σταματάμε το 1ο, το συνεχίζουμε και τέλος τερματίζουμε και τις δυο διεργασίες.

```
$ ./a.exe
Process 1 is alive
Process 2 is alive
Process 1 is alive
Process 2 is alive
Process 1 is alive
Process 2 is alive
```

Ζ. Κλήση Συστήματος alarm()

#include <unistd.h>

unsigned int alarm(unsigned int count);

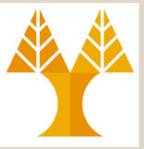
Επιστρέφει 0 ή τον αριθμό δευτερολ. μέχρι το προηγούμενο set alarm.

- **Δίνει την εντολή στον πυρήνα να στείλει μετά από count δευτερόλεπτα το σήμα SIGALRM στην καλούσα διεργασία.**
- Ο μετρητής κρατείται μέσα στον πυρήνα.

Επισημάνσεις

- **Το alarm ΔΕΝ κληρονομείται στο παιδί.**
- Επίσης το alarm γίνεται reset κάθε φορά που φθάνει στην διεργασία οπόταν πρέπει να ξαναγίνει installed εάν θέλουμε να το λαμβάνουμε συνέχεια (δες παράδειγμα)

7. Παράδειγμα alarm()



```
#include <signal.h>  // signal, alarm
#include <unistd.h>  // fork, getpid
#include <time.h>    // time_t, time

void signal_handler(int sig) {
   time_t rawtime;    time ( &rawtime );  /* compute the current time */
   printf("#%d: The kernel sent a SIGALRM(%d) on: %s", getpid(), sig, ctime (&rawtime));
   alarm(1);  /* we re-install the alarm here */
}

int main() {
        int pid;
        /* Install signal handler and set the alarm clock */
        signal(SIGALRM, signal_handler);
        alarm(1);
        pid = fork();
        if (pid == -1) {    perror("fork");  exit(1); }
        else if (pid == 0) {              /* The child process */
                while (1)  { printf("#%d\n", getpid()); sleep(3); }
        }
        else {
                while (1) { printf("#%d\n", getpid()); sleep(3); }
        }
}
```

Alarm signal handler

Αποτέλεσμα Εκτέλεσης

```
#6020 (child)
#2280 (parent)
#2280: The kernel sent a SIGALRM(14) on: Sun Mar  4 23:58:58 2007
#2280
#2280: The kernel sent a SIGALRM(14) on: Sun Mar  4 23:58:59 2007
#2280
#2280: The kernel sent a SIGALRM(14) on: Sun Mar  4 23:59:00 2007
#2280
#6020
```

Μόνο ο πατέρας παραλαμβάνει το alarm