ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΑΚΑΔΗΜΑΪΚΟ ΕΤΟΣ 2016-2017

ΧΕΙΜΕΡΙΝΟ ΕΞΑΜΗΝΟ

ΜΑΘΗΜΑ: ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

ΕΡΓΑΣΤΗΡΙΟ 4 - ΣΗΜΕΙΩΣΕΙΣ

Εισαγωγή στη Διαχείριση Διεργασιών

Όταν εκκινεί το Λειτουργικό Σύστημα, ένα σύνολο λειτουργιών εκκινούν ως ξεχωριστές διεργασίες και καλεί ένα πρόγραμμα το οποίο ονομάζεται ***init***. Επίσης, ένας αριθμός από scripts τα οποία βρίσκονται στον κατάλογο **/etc** (init scripts) αναλαμβάνουν να εκτελέσουν τις λειτουργίες του συστήματος. Πολλά από τα προγράμματα του Λειτουργικού Συστήματος εκτελούνται ως 'δαίμονες' (daemons) τα οποία εκκινούν και εκτελούνται στο background, συνήθως, χωρίς αλληλεπίδραση με τους χρήστες.

Στο Linux μπορεί να δημιουργηθεί μια ιεραρχία διεργασιών με τη μορφή πατέρας – παιδί (parent – child). Ο πυρήνας κρατά συγκεκριμένες πληροφορίες για κάθε διεργασία όπως το ID κάθε διεργασίας (process ID – PID). Τα PIDs αποδίδονται σε αύξουσα σειρά με τη διεργασία init να έχει πάντα PID = 1. Επίσης, ο πυρήνας κρατά τη μνήμη που αποδίδεται σε κάθε διεργασία όπως και την 'ετοιμότητα' κάθε διεργασίας να συνεχίσει την εκτέλεσή της.

Για να δούμε πληροφορίες που σχετίζονται με τις διεργασίες που εκτελούνται στο Λειτουργικό Σύστημα μπορούμε να χρησιμοποιήσουμε την εντολή **ps**. Στο ακόλουθο παράδειγμα, η ps μας δείχνει ότι εκτελούνται δύο διεργασίες: ο φλοιός bash και η ίδια η εντολή ps.

```
kk@ubuntu:~$ ps
  PID TTY          TIME CMD
 2487 pts/0    00:00:00 bash
 3237 pts/0    00:00:00 ps
```

Η εντολή **ps x** μας δείχνει περισσότερες πληροφορίες σχετικά με τις διεργασίες που εκτελούνται από το Λειτουργικό σύστημα. Ο χαρακτήρας ? στη στήλη TTY μας λέει ότι οι συγκεκριμένες διεργασίες δεν έχουν ένα συγκεκριμένο τερματικό ελέγχου. Στη στήλη STAT μπορούμε να διακρίνουμε την κατάσταση κάθε διεργασίας. Ο επόμενος πίνακας συνοψίζει τις πιθανές καταστάσεις των διεργασιών.

STAT Code	Description
S	Sleeping. Usually waiting for an event to occur, such as a signal or input to become available.
R	Running. Strictly speaking, "runnable," that is, on the run queue either executing or about to run.
D	Uninterruptible Sleep (Waiting). Usually waiting for input or output to complete.
T	Stopped. Usually stopped by shell job control or the process is under the control of a debugger.
Z	Defunct or "zombie" process.
N	Low priority task, "nice."
W	Paging. (Not for Linux kernel 2.6 onwards.)
s	Process is a session leader.
+	Process is in the foreground process group.
l	Process is multithreaded.
<	High priority task.

Οι χρήστες που ‘τρέχουν’ τις διεργασίες παρουσιάζονται στα αποτελέσματα της εντολής **ps aux**. Ο ακόλουθος πίνακας παρουσιάζει τη σημασία της κάθε στήλης αποτελεσμάτων της ps aux.

Header	Meaning
USER	User ID. This is the owner of the process.
%CPU	CPU usage as a percent.
%MEM	Memory usage as a percent.
VSZ	Virtual memory size.
RSS	Resident Set Size. The amount of physical memory (RAM) the process is using in kilobytes.
START	Time when the process started. For values over 24 hours, a date is used.

Για τη διαχείριση διεργασιών χρησιμοποιούνται οι ακόλουθες εντολές:.

- **Εντολή top**

Η εντολή top παρουσιάζει πληροφορίες σχετικά με τις διεργασίες που εκτελούνται. Οι πληροφορίες ανανεώνονται κάθε 3 δευτερόλεπτα. Στην αρχή των αποτελεσμάτων παρουσιάζονται οι συνολικές πληροφορίες του συστήματος ενώ η λίστα των διεργασιών ξεκινά με αυτές τις διεργασίες που έχουν τη μεγαλύτερη δραστηριότητα. Ο ακόλουθος πίνακας παρουσιάζει τη σημασιολογία των πληροφοριών που εμφανίζει η εντολή top.

```
top - 14:59:20 up  6:30,  2 users,  load average: 0.07, 0.02, 0.00
Tasks: 109 total,   1 running, 106 sleeping,   0 stopped,   2 zombie
Cpu(s):  0.7%us,  1.0%sy,  0.0%ni, 98.3%id,  0.0%wa,  0.0%hi,  0.0%si
Mem:    319496k total,   314860k used,     4636k free,    19392k buff
Swap:   875500k total,   149128k used,   726372k free,   114676k cach

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 6244 me        39  19 31752 3124 2188 S  6.3  1.0  16:24.42 trackerd
11071 me        20   0  2304 1092  840 R  1.3  0.3   0:00.14 top
 6180 me        20   0  2700 1100  772 S  0.7  0.3   0:03.66 dbus-dae
 6321 me        20   0 20944 7248 6560 S  0.7  2.3   2:51.38 multiloa
 4955 root      20   0  104m 9668 5776 S  0.3  3.0   2:19.39 Xorg
    1 root      20   0  2976  528  476 S  0.0  0.2   0:03.14 init
    2 root      15  -5     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 migratio
    4 root      15  -5     0    0    0 S  0.0  0.0   0:00.72 ksoftirq
    5 root      RT  -5     0    0    0 S  0.0  0.0   0:00.04 watchdog
    6 root      15  -5     0    0    0 S  0.0  0.0   0:00.42 events/0
    7 root      15  -5     0    0    0 S  0.0  0.0   0:00.06 khelper
   41 root      15  -5     0    0    0 S  0.0  0.0   0:01.08 kblockd/
   67 root      15  -5     0    0    0 S  0.0  0.0   0:00.00 kseriod
  114 root      20   0     0    0    0 S  0.0  0.0   0:01.62 pdflush
  116 root      15  -5     0    0    0 S  0.0  0.0   0:02.44 kswapd0
```

Row	Field	Meaning
1	`top`	Name of the program.
	`14:59:20`	Current time of day.
	`up 6:30`	This is called *uptime*. It is the amount of time since the machine was last booted. In this example, the system has been up for 6½ hours.
	`2 users`	Two users are logged in.
	`load average:`	*Load average* refers to the number of processes that are waiting to run; that is, the number of processes that are in a runnable state and are sharing the CPU. Three values are shown, each for a different period of time. The first is the average for the last 60 seconds, the next the previous 5 minutes, and finally the previous 15 minutes. Values under 1.0 indicate that the machine is not busy.
2	`Tasks:`	This summarizes the number of processes and their various process states.
	`0.7%us`	0.7% of the CPU is being used for *user processes*. This means processes outside of the kernel itself.
	`1.0%sy`	1.0% of the CPU is being used for *system* (kernel) processes.
	`0.0%ni`	0.0% of the CPU is being used by nice (low-priority) processes.
	`98.3%id`	98.3% of the CPU is idle.
	`0.0%wa`	0.0% of the CPU is waiting for I/O.
4	`Mem:`	Shows how physical RAM is being used.
5	`Swap:`	Shows how swap space (virtual memory) is being used.

- **Χαρακτήρας &**

Ο χαρακτήρας & χρησιμοποιείται για να θέσει την εκτέλεση μιας διεργασίας στο παρασκήνιο (background). Για παράδειγμα, δώστε την εντολή ***xlogo &*** και έπειτα την εντολή ***ps***.

- **Εντολή jobs**

Η εντολή jobs μας εμφανίζει τις διεργασίες που τρέχουν.

- **Εντολή fg**

Αφού δώσουμε την εντολή jobs, με την εντολή ***fg %<number>*** μπορούμε να επαναφέρουμε μια διεργασία στο προσκήνιο (foreground). Για να σταματήσουμε μια διεργασία που τρέχει στο προσκήνιο μπορούμε να πιέσουμε Ctrl-Z.

- **Εντολή bg**

Με την εντολή bg θέτουμε μια διεργασία για εκτέλεση στο παρασκήνιο (background).

- **Εντολή kill**

Με την εντολή kill τερματίζουμε μια διεργασία. Με την εντολή μπορούμε να στείλουμε και σήματα στις διεργασίες ως εξής: **kill <-signal> PID**. Αν δεν καθοριστεί κάποιο σήμα, τότε στέλνεται το σήμα TERM (terminate). Ο ακόλουθος πίνακας παρουσιάζει τα πιο συνηθισμένα σήματα:

Type	Name	Number	Generating condition
Not a real signal	EXIT	0	Exit because of exit command or reaching the end of the program (not an actual signal but useful in trap)
Hang up	SIGHUP or HUP	1	Disconnect the line
Terminal interrupt	SIGINT or INT	2	Press the interrupt key (usually CONTROL-C)
Quit	SIGQUIT or QUIT	3	Press the quit key (usually CONTROL-SHIFT-| or CONTROL-SHIFT-\)
Kill	SIGKILL or KILL	9	The kill builtin with the **–9** option (cannot be trapped; use only as a last resort)
Software termination	SIGTERM or TERM	15	Default of the kill command
Stop	SIGTSTP or TSTP	20	Press the suspend key (usually CONTROL-Z)
Debug	DEBUG		Executes ***commands*** specified in the trap statement after each command (not an actual signal but useful in trap)
Error	ERR		Executes ***commands*** specified in the trap statement after each command that returns a nonzero exit status (not an actual signal but useful in trap)

Η αναφορά στα σήματα μπορεί να γίνει είτε αριθμητικά είτε ονομαστικά χρησιμοποιώντας τα ονόματα που παρουσιάζονται στον παραπάνω πίνακα (προβάλλοντας τους χαρακτήρες SIG).
Παραδείγματα:
$ xlogo &
$ kill -INT 13601
[1]+ Interrupt xlogo
$ xlogo &
[1] 13608
$ kill -SIGINT 13608
[1]+ Interrupt xlogo

Άλλα είδη σημάτων είναι τα ακόλουθα:

Number	Name	Meaning
3	QUIT	Quit.
11	SEGV	Segmentation violation. This signal is sent if a program makes illegal use of memory; that is, it tried to write somewhere it was not allowed to.
20	TSTP	Terminal stop. This is the signal sent by the terminal when CTRL-Z is pressed. Unlike the STOP signal, the TSTP signal is received by the program but the program may choose to ignore it.
28	WINCH	Window change. This is a signal sent by the system when a window changes size. Some programs, like top and less, will respond to this signal by redrawing themselves to fit the new window dimensions.

Ο τερματισμός ή η αποστολή σημάτων σε πολλαπλές διεργασίες μπορεί να γίνει με τη χρήση της εντολής **killall <name>**.

Άλλες εντολές που αφορούν σε διεργασίες είναι οι ακόλουθες:

Command	Description
`pstree`	Outputs a process list arranged in a tree-like pattern showing the parent/child relationships between processes.
`vmstat`	Outputs a snapshot of system resource usage including memory, swap, and disk I/O. To see a continuous display, follow the command with a time delay (in seconds) for updates (e.g., `vmstat 5`). Terminate the output with CTRL-C.
`xload`	A graphical program that draws a graph showing system load over time.
`tload`	Similar to the `xload` program, but draws the graph in the terminal. Terminate the output with CTRL-C.

Γενικά, όταν μια διεργασία δημιουργεί / καλεί μια νέα διεργασία τότε ονομάζεται **πατρική διεργασία** (parent) ενώ η καλούμενη ονομάζεται **διεργασία παιδί** (child). Σε ένα υπολογιστικό σύστημα πολλαπλές διεργασίες προσπαθούν να καταλάβουν τον επεξεργαστή αφού μόνο μια κάθε φορά είναι αυτή που εκτελείται. Ο πυρήνας του Linux χρησιμοποιεί ένα scheduler διεργασιών ώστε να αποφασίζει το ποια διεργασία θα είναι αυτή που θα χρησιμοποιήσει τον επεξεργαστή. Κάθε διεργασία έχει και μια προτεραιότητα που υιοθετείται από τον scheduler ώστε να αποφασίσει τη σειρά εκτέλεσης των διεργασιών. ***Διεργασίες με υψηλή προτεραιότητα εκτελούνται πιο συχνά ενώ διεργασίες με χαμηλή προτεραιότητα εκτελούνται πιο αραιά. Η προκαθορισμένη προτεραιότητα είναι ίση με 0. Το εύρος τιμών των προτεραιοτήτων είναι από -20 μέχρι +20. Αρνητικές τιμές δείχνουν υψηλή προτεραιότητα σε σχέση με τις θετικές τιμές.***

Το λειτουργικό σύστημα καθορίζει την προτεραιότητα κάθε διεργασίας με βάση μια nice τιμή και την συμπεριφορά της κάθε διεργασίας. Για παράδειγμα, διεργασίες που εκτελούνται για μεγάλες χρονικές περιόδους χωρίς διακοπές 'παίρνουν' χαμηλές προτεραιότητες. Από την άλλη, διεργασίες που διακόπτονται, π.χ. για να πάρουν είσοδο πριμοδοτούνται. Η εντολή nice μπορεί να χρησιμοποιηθεί ώστε να αυξήσει την τιμή της προτεραιότητας μιας διεργασίας και με αυτό τον τρόπο να μειώσει την προτεραιότητα εκτέλεσής της.

Στην γραμμή εντολών δίνουμε:

$ ps –l

και μας παρουσιάζονται πληροφορίες σχετικά με τις διεργασίες ανάμεσα στις οποίες και η προτεραιότητά τους. Εκτελώντας μια διεργασία στο παρασκήνιο, π.χ.

$ nice xlogo &

η διεργασία xlogo εκτελείται στο παρασκήνιο με την προκαθορισμένη προτεραιότητα. Με την εντολή:

$ renice 10 <PID>

μπορούμε να θέσουμε την προτεραιότητα της διεργασίας με το <PID> στο 10 και να μειώσουμε την προτεραιότητα εκτέλεσης της.

Δημιουργία και Εκτέλεση Διεργασιών

Γενικά, έχουμε τη δυνατότητα μέσα από ένα πρόγραμμα να δημιουργήσουμε μια διεργασία και να την εκτελέσουμε με τη βοήθεια της βιβλιοθήκης stdlib.h της C. Για το σκοπό αυτό κάνουμε χρήση της συνάρτησης

int system (const char *string);

Στο ακόλουθο παράδειγμα, παρουσιάζουμε την εκτέλεση της εντολής ps ax του Linux μέσα από ένα πρόγραμμα C.

Code 1

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
        printf("Running ps with system\n");
        system("ps ax");
        printf("Done.\n");
        exit(0);
}
```

Μπορούμε να θέσουμε την εκτέλεση της εντολής ps ax στο παρασκήνιο θέτοντας την εντολή system("ps ax &");.

Προγραμματιστικά το PID αναπαριστάται από τον τύπο pid_t που ορίζεται στη βιβλιοθήκη <sys/types.h>. Η κλήση συστήματος pid_t getpid (void); μας επιστρέφει το PID της διεργασίας ενώ η pid_t getppid (void); μας δίνει το PID της πατρικής. Το μέγιστο PID είναι ίσο με 32768. Το ακόλουθο πρόγραμμα αποτελεί ένα χαρακτηριστικό παράδειγμα.

Code 2

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

Τερματισμός Διεργασιών

Ο τερματισμός των διεργασιών γίνεται με την εντολή void exit (int status); και με χρήση της βιβλιοθήκης <stdlib.h>. Μια κλήση στην exit() οδηγεί στην εκτέλεση κάποιων απαραίτητων βημάτων και στη συνέχεια στον τερματισμό της διεργασίας. Το status & 0377 επιστρέφει στην πατρική διεργασία, η τιμή 0 δείχνει επιτυχία ενώ μια τιμή διαφορετική του 0 δείχνει ανεπιτυχή εκτέλεση. Τα EXIT_SUCCESS και EXIT_FAILURE καθορίζονται για να δείξουν επιτυχία ή αποτυχία. Συνεπώς, μια επιτυχής έξοδος μπορεί να επιδειχθεί με την ακόλουθη εντολή exit (EXIT_SUCCESS);. Τα βήματα που επιτελούνται πριν το τερματισμό μιας διεργασίας είναι τα ακόλουθα:

- Καλούνται όλες οι συναρτήσεις που έχουν καταχωρηεθεί με τις atexit( ), on_exit( ) σε αντίστροφη σειρά καταχώρησης.
- Επιχειρείται το flushing όλων των ανοιχτών I/O ροών.
- Διαγράφονται όλα τα προσωρινά αρχεία που έχουν δημιουργηθεί με την συνάρτηση tmpfile( ).

Η exit() καλεί την _exit() ώστε ο πυρήνας να διαχειριστεί την υπόλοιπη διαδικασία τερματισμού της διεργασίας.

#include <unistd.h>

void _exit (int status);

Συνάρτηση atexit()

Η κλήση συστήματος atexit() επιτρέπει τον καθορισμό συναρτήσεων που θα εκτελεστούν κατά την ομαλή διαδικασία τερματισμού μιας διεργασίας, π.χ. με κλήση της exit() ή της return.

#include <stdlib.h>

int atexit (void (*function)(void));

Παράδειγμα:

Code 3

```
#include <stdio.h>
#include <stdlib.h>
void out (void)
{
        printf ("atexit( ) succeeded!\n");
}
int main (void)
{
        if (atexit (out))
        fprintf(stderr, "atexit( ) failed!\n");
        return 0;
}
```

Συνάρτηση on_exit

Ενώ η atexit() έχει καθοριστεί σύμφωνα με το POSIX 1003.1-2001, το SunOS 4 ορίζει την ισοδύναμη της που είναι η on_exit(). Η on_exit() έχει την ίδια λειτουργία με την atexit().

#include <stdlib.h>

int on_exit (void (*function)(int , void *), void *arg);

Οικογένεια Εντολών exec

Η οικογένεια εντολών exec μας επιτρέπει να εκκινήσουμε νέες διεργασίες. Κάθε έκδοση της exec διαφοροποιείται στον τρόπο που εκκινεί τις διεργασίες και παρουσιάζει τις παραμέτρους των προγραμμάτων. Η exec αντικαθιστά την τρέχουσα διεργασία με μια νέα διεργασία όπως καθορίζεται από τις παραμέτρους της εντολής. Με την exec διευκολύνεται το πέρασμα από το ένα πρόγραμμα σε κάποιο άλλο. Οι εντολές exec έχουν ως εξής:

#include <unistd.h>

char **environ;

int execl(const char *path, const char *arg0, ..., (char *)0);

int execlp(const char *file, const char *arg0, ..., (char *)0);

int execle(const char *path, const char *arg0, ..., (char *)0, char *const envp[]);

int execv(const char *path, char *const argv[]);

int execvp(const char *file, char *const argv[]);

int execve(const char *path, char *const argv[], char *const envp[]);

Το παρακάτω τμήμα προγράμματος αποτελεί ένα παράδειγμα κλήσης των εντολών exec για την εκτέλεση της εντολής ps.

```
#include <unistd.h>

/* Example of an argument list */
/* Note that we need a program name for argv[0] */
char *const ps_argv[] =
    {"ps", "ax", 0};

/* Example environment, not terribly useful */
char *const ps_envp[] =
    {"PATH=/bin:/usr/bin", "TERM=console", 0};

/* Possible calls to exec functions */
execl("/bin/ps", "ps", "ax", 0);            /* assumes ps is in /bin */
execlp("ps", "ps", "ax", 0);                /* assumes /bin is in PATH */
execle("/bin/ps", "ps", "ax", 0, ps_envp);  /* passes own environment */

execv("/bin/ps", ps_argv);
execvp("ps", ps_argv);
execve("/bin/ps", ps_argv, ps_envp);
```

Code 4

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
        printf("Running ps with execlp\n");
        execlp("ps", "ps", "ax", (char *)0);
        execl("/bin/ps", "ps", "ax", (char *)0);
        printf("Done.\n");
        exit(0);
}
```

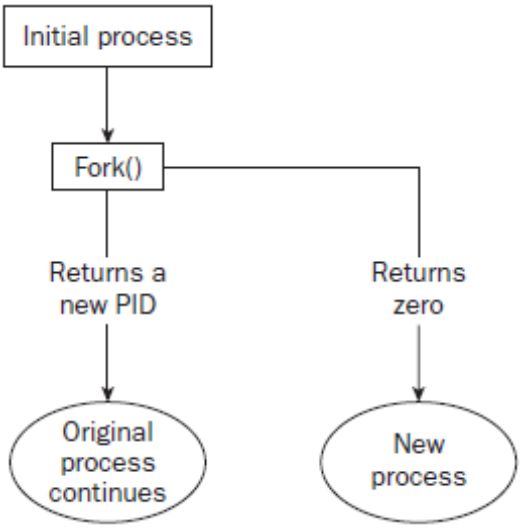
Εντολή fork

Όταν επιθυμούμε να χρησιμοποιήσουμε τις διεργασίες ώστε να εκτελούν περισσότερες από μια εργασίες κάθε φορά τότε θα πρέπει είτε να υιοθετήσουμε προγραμματισμό με νήματα (threads) είτε να δημιουργήσουμε μια ξεχωριστή διεργασία για το σκοπό αυτό. Για τη δημιουργία ξεχωριστών διεργασιών χρησιμοποιούμε την εντολή **fork()**. Η εντολή 'διπλασιάζει' την τρέχουσα διεργασία δημιουργώντας μια νέα εγγραφή στον πίνακα των διεργασιών που διατηρεί το Λειτουργικό Σύστημα. Η νέα διεργασία είναι σχεδόν πανομοιότυπη με την αρχική και εκτελεί τον ίδιο κώδικα όμως με το δικό της χώρο δεδομένων, περιβάλλον και περιγραφείς αρχείων. Η νέα διεργασία καλείται διεργασία – παιδί (child) ενώ η αρχική καλείται διεργασία – πατέρας (parent). Η οικογένεια εντολών exec και η fork() σας προσφέρουν ένα σύνολο δυνατοτήτων για τη δημιουργία νέων διεργασιών. Η σύνταξη της fork() έχει ως ακολούθως:

#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);

Το επόμενο σχήμα μας δείχνει τον τρόπο εκτέλεσης της fork().



Η fork() μας επιτρέπει να αναγνωρίσουμε μέσα στον κώδικα το ποια διεργασία είναι μέσω του αποτελέσματος εκτέλεσης. **Αν η fork() επιστρέψει -1**, τότε κάποιο σφάλμα έχει συμβεί. Για παράδειγμα, το σφάλμα εκτέλεσης θα εμφανιστεί αν η πατρική διεργασία ξεπεράσει το μέγιστο αριθμό παιδιών που μπορεί να δημιουργήσει (CHILD_MAX). Επίσης, μπορεί να μην υπάρχει αρκετή μνήμη στο σύστημα ώστε να υποστηριχθεί η δημιουργία της νέας διεργασίας. **Αν η fork() επιστρέψει 0**, τότε πρόκειται για τη διεργασία παιδί ενώ στην πατρική διεργασία επιστρέφει το PID της νέας διεργασίας – παιδί. Η νέα διεργασία και η πατρική συνεχίζουν να εκτελούνται όπως η αρχική διεργασία. Τα επόμενα τμήματα κώδικα μας δίνουν κάποια παραδείγματα χειρισμού της fork() και των διεργασιών που δημιουργεί.

Παράδειγμα 1

```
pid_t new_pid;
new_pid = fork();
switch(new_pid) {
        case -1 : /* Error */
                break;
        case 0 : /* We are child */
                break;
        default : /* We are parent */
                break;
}
```

Παράδειγμα 2

```
pid_t pid;
pid = fork ( );
if (pid > 0)
        printf ("I am the parent of pid=%d!\n", pid);
else if (!pid)
        printf ("I am the baby!\n");
else if (pid == -1)
        perror ("fork");
```

Το επόμενο παράδειγμα χρησιμοποιεί την fork() ώστε να δημιουργήσει μια διεργασία – παιδί και στη συνέχεια και οι δύο, πατρική διεργασία και διεργασία παιδί εκτυπώνουν κάποια μηνύματα στην οθόνη.

Code 5

```
#include <sys/types.h>
#include <unistd.h>
```

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        pid_t pid;
        char *message;
        int n;
        printf("fork program starting\n");
        pid = fork();
        switch(pid)
        {
                case -1:
                        perror("fork failed");
                        exit(1);
                case 0:
                        message = "This is the child";
                        n = 5;
                        break;
                default:
                        message = "This is the parent";
                        n = 3;
                        break;
        }
        for(; n > 0; n--) {
                puts(message);
                sleep(1);
        }
        exit(0);
}
```

Ο ακόλουθος κώδικας αποτελεί ένα ακόμα παράδειγμα χρήσης της fork().

Code 6

```
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>

int main()
{
   pid_t childpid;
   int retval;
   int status;

   childpid = fork();

   if (childpid >= 0)
   {
      if (childpid == 0)
      {
         printf("CHILD: I am the child process!\n");
         printf("CHILD: Here's my PID: %d\n", getpid());
         printf("CHILD: My parent's PID is: %d\n", getppid());
```

```
            printf("CHILD: The value of my copy of childpid is: %d\n", childpid);
            printf("CHILD: Sleeping for 1 second...\n");
            sleep(1);
            printf("CHILD: Enter an exit value (0 to 255): ");
            scanf(" %d", &retval);
            printf("CHILD: Goodbye!\n");
            exit(retval);
        }
        else
        {
            printf("PARENT: I am the parent process!\n");
            printf("PARENT: Here's my PID: %d\n", getpid());
            printf("PARENT: The value of my copy of childpid is %d\n", childpid);
            printf("PARENT: I will now wait for my child to exit.\n");
            wait(&status); /* wait for child to exit, and store its status */
            printf("PARENT: Child's exit code is: %d\n", WEXITSTATUS(status));
            printf("PARENT: Goodbye!\n");
            exit(0);
        }
    }
    else
    {
        perror("fork");
        exit(0);
    }
}
```

Αναμονή Τερματισμού Διεργασιών

Η δημιουργία μιας νέας διεργασίας με την εντολή fork() αποδίδει ανεξαρτησία στην εκτέλεση της διεργασίας – παιδί. Όμως, ορισμένες φορές, η πατρική διεργασία θα πρέπει να γνωρίζει το πέρας της εκτέλεσης της διεργασίας – παιδί ιδιαίτερα σε περιπτώσεις όπου η πατρική – διεργασία δημιουργεί πολλαπλές διεργασίες ώστε να εκτελέσουν μια συγκεκριμένη λειτουργία. Η εντολή wait επιτρέπει στην πατρική διεργασία να περιμένει το πέρας της εκτέλεσης των διεργασιών – παιδιών. Η σύνταξη της εντολής wait έχει ως εξής:

#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *stat_loc);

Με την εντολή wait η διεργασία – πατέρας σταματά μέχρι ένα από τις διεργασίες – παιδιά τερματίσει την εκτέλεσή του. Η κλήση της εντολής επιστρέφει το PID διεργασίας – παιδί. Η πληροφορία της κατάστασης της διεργασίας παιδιού (status information) είναι η τιμή που επιστρέφει η main ή η τιμή που περνάει με την εντολή exit. Εφόσον ο δείκτης stat_loc δεν είναι null, στη θέση που δείχνει ο δείκτης θα γραφτεί το status information. Στη βιβλιοθήκη sys/wait.h περιλαμβάνεται ένα σύνολο μακρο-εντολών με τις οποίες μπορούμε να μεταφράσουμε το status information. Ο επόμενος πίνακας περιλαμβάνει τις μακρο-εντολές αυτές:

Macro	Definition
WIFEXITED(stat_val)	Nonzero if the child is terminated normally.
WEXITSTATUS(stat_val)	If WIFEXITED is nonzero, this returns child exit code.
WIFSIGNALED(stat_val)	Nonzero if the child is terminated on an uncaught signal.
WTERMSIG(stat_val)	If WIFSIGNALED is nonzero, this returns a signal number.
WIFSTOPPED(stat_val)	Nonzero if the child has stopped.
WSTOPSIG(stat_val)	If WIFSTOPPED is nonzero, this returns a signal number.

Το ακόλουθο τμήμα κώδικα αποτελεί ένα παράδειγμα χρήσης της εντολής wait() και συγχρονισμού διεργασιών.

Code 7

```
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

#include <stdlib.h>
int main()
{
        pid_t pid;
        char *message;
        int n;
        int exit_code;
        printf("fork program starting\n");
        pid = fork();
        switch(pid)
        {
                case -1:
                        perror("fork failed");
                        exit(1);
                case 0:
                        message = "This is the child";
                        n = 5;
                        exit_code = 37;
                        break;
                default:
                        message = "This is the parent";
                        n = 3;
                        exit_code = 0;
                        break;
        }
        for(; n > 0; n--) {
                puts(message);
                sleep(1);
        }

        if (pid != 0) {
                int stat_val;
                pid_t child_pid;
                child_pid = wait(&stat_val);
                printf("Child has finished: PID = %d\n", child_pid);
```

```
		if(WIFEXITED(stat_val))
			printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
		else
			printf("Child terminated abnormally\n");
	}
	exit(exit_code);
}
```

Όταν μια διεργασία – παιδί τερματίσει τότε η σύνδεση πατέρα – παιδί παραμένει ‘ζωντανή’ μέχρι η διεργασία - πατέρας να τερματίσει ή να καλέσει ξανά την εντολή wait. Για αυτό το λόγο, η διεργασία παιδί παραμένει στον πίνακα των διεργασιών αφού ο κωδικός εξόδου της πρέπει να είναι αποθηκευμένος αφού υπάρχει η πιθανότητα η διεργασία – πατέρας να καλέσει την εντολή wait. Σε αυτές τις περιπτώσεις η διεργασία – παιδί ονομάζεται διεργασία zombie. Στο ακόλουθο παράδειγμα, η διεργασία – παιδί τερματίζει την εκτύπωση των μηνυμάτων πριν την πατρική διεργασία. Για να δείτε τη διεργασία zombie, εκτελέστε το πρόγραμμα στο παρασκήνιο και δώστε την εντολή **ps – al**. Η διεργασία zombie θα έχει την ένδειξη **defunct**.

Code 8

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

#include <stdlib.h>
int main()
{
	pid_t pid;
	char *message;
	int n;
	int exit_code;
	printf("fork program starting\n");
	pid = fork();
	switch(pid)
	{
		case -1:
			perror("fork failed");
			exit(1);
		case 0:
			message = "This is the child";
			n = 3;
			break;
		default:
			message = "This is the parent";
			n = 15;
			break;
	}

	for(; n > 0; n--) {
		puts(message);
		sleep(2);
	}

	if (pid != 0) {
		int stat_val;
```

```
                pid_t child_pid;
                child_pid = wait(&stat_val);
                printf("Child has finished: PID = %d\n", child_pid);
                if(WIFEXITED(stat_val))
                        printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
                else
                        printf("Child terminated abnormally\n");
        }
        exit(exit_code);
}
```

Τέλος, ένας άλλος τρόπος για να περιμένουμε τον τερματισμό εκτέλεσης μιας διεργασίας – παιδί είναι η εντολή waitpid(). Η σύνταξη της έχει ως εξής:

#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *stat_loc, int options);

Η παράμετρος pid καθορίζει επ' ακριβώς το ποια διεργασία είναι αυτή της οποίας θα αναμείνουμε τον τερματισμό. Οι τιμές που μπορεί να πάρει έχουν ως ακολούθως:

- < -1. Αναμένει για οποιαδήποτε διεργασία – παιδί για την οποία το ID της ομάδας διεργασιών είναι ίσο με την απόλυτη τιμή της τιμής που δίνουμε. Για παράδειγμα, αν βάλουμε -500 περιμένουμε να τερματιστεί η ομάδα διεργασιών με ID ίσο με 500.
- -1. Αναμένουμε για τον τερματισμό οποιασδήποτε διεργασίας – παιδί (ίδια με την εντολή wait().
- 0. Αναμένουμε για τον τερματισμό οποιασδήποτε διεργασίας – παιδί που ανήκει στην ίδια ομάδα διεργασιών.
- > 0. Αναμένουμε τον τερματισμό οποιαδήποτε διεργασίας που ανήκει στην ομάδα διεργασιών που έχει ID ίσο με τον αριθμό που δίνουμε.

Ο ακόλουθος πίνακας παρουσιάζει τα options που μπορούμε να χρησιμοποιήσουμε στην εντολή waitpid().

Option	**Σημασιολογία**
WNOHANG	Δεν μπλοκάρει αλλά επιστρέφει άμεσα όταν καμία διεργασία – παιδί δεν έχει τερματίσει.
WUNTRACED	Αν οριστεί, ορίζεται επίσης η WIFSTOPPED ακόμα και αν η διεργασία που την καλεί δεν εντοπίζει τη διεργασία – παιδί.
WCONTINUED	Αν οριστεί, ορίζεται και η WIFCONTINUED και αν η διεργασία που την καλεί δεν εντοπίζει τη διεργασία – παιδί.

Δαίμονες (Daemons)

Ένας δαίμονας (daemon) είναι μια διεργασία που τρέχει στο παρασκήνιο χωρίς να έχει συνδεθεί σε κάποιο συγκεκριμένο τερματικό ελέγχου. Συνήθως εκκινούν μαζί με την εκκίνηση του Λειτουργικού Συστήματος και αναλαμβάνουν εργασίες ελέγχου του συστήματος. Γενικά τα βήματα για να δημιουργήσουμε μια διεργασία – δαίμονα είναι τα ακόλουθα:

1. Καλούμε την εντολή fork().
2. Στην πατρική διεργασία εκτελούμε την εντολή exit. Αυτό εξασφαλίζει η αρχική διεργασία – παιδί ικανοποιείται με τον τερματισμό της διεργασίας – παιδί, ότι η

πατρική διεργασία του δαίμονα δεν εκτελείται πλέον και ότι η διεργασία δαίμονας δεν αποτελεί μια διεργασία πατέρα μιας ομάδας διεργασιών.

3. Καλούμε την εντολή setsid() δίνοντας στη διεργασία δαίμονα μια νέα ομάδα διεργασιών και σύνοδο.
4. Αλλάζουμε τον τρέχοντα κατάλογο στο root ώστε να εξασφαλίσουμε ότι δεν βρισκόμαστε οπουδήποτε στη δομή αρχείων – καταλόγων.
5. Κλείνουμε όλους τους περιγραφείς αρχείων.
6. Ανοίγουμε μόνο τους περιγραφείς αρχείων 0, 1 και 2 (standard input, standard output, standard error).

Η κλήση setsid() δημιουργεί μια νέα σύνοδο (session) υποθέτωντας ότι η διεργασία που την καλεί δεν είναι ήδη μια διεργασία που ελέγχει μια ομάδα διεργασιών. Η σύνταξη της κλήσης έχει ως εξής:

#include <unistd.h>
pid_t setsid (void);

Η διεργασία που κάνει την κλήση γίνεται ο leader της συνόδου η οποία δεν έχει κάποιο συγκεκριμένο τερματικό ελέγχου. Η κλήση επίσης δημιουργεί μια νέα ομάδα διεργασιών και τοποθετεί ως μόνο μέλος τη διεργασία που έκανε την κλήση. Κατά την επιτυχή εκτέλεση κλήση επιστρέφει το session ID. Σε σφάλμα επιστρέφει -1. Ο πιο εύκολος τρόπος για να εξασφαλίσουμε ότι οποιαδήποτε διεργασία δεν είναι ο leader μιας ομάδας διεργασιών είναι να ετκελέσουμε την εντολή fork(), να τερματίσουμε την πατρική διεργασία και να καλέσουμε την setsid() στη διεργασία – παιδί. Για παράδειγμα:

```
pid_t pid;
pid = fork ( );
if (pid == -1) {
        perror ("fork");
        return -1;
}  else if (pid != 0)
        exit (EXIT_SUCCESS);
if (setsid ( ) == -1) {
        perror ("setsid");
        return -1;
}
```

Στο ακόλουθο πρόγραμμα δημιουργούμε μια διεργασία – δαίμονα και εφαρμόζουμε τα παραπάνω βήματα.

Code 9

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
int main(int argc, char* argv[])
{
        FILE *fp= NULL;
        pid_t process_id = 0;
        pid_t sid = 0;

        // Create child process
        process_id = fork();
        // Indication of fork() failure
        if (process_id < 0)
```

```
        {
                printf("fork failed!\n");
                // Return failure in exit status
                exit(1);
        }
        // PARENT PROCESS. Need to kill it.
        if (process_id > 0)
        {
                printf("process_id of child process %d \n", process_id);
                // return success in exit status
                exit(0);
        }
        //unmask the file mode
        umask(0);
        //set new session
        sid = setsid();
        if(sid < 0)
        {
                // Return failure
                exit(1);
        }
        // Change the current working directory to root.
        chdir("/");
        // Close stdin. stdout and stderr
        close(STDIN_FILENO);
        close(STDOUT_FILENO);
        close(STDERR_FILENO);
        // Open a log file in write mode.
        fp = fopen ("Log.txt", "w+");
        while (1)
        {
                //Dont block context switches, let the process sleep for some time
                sleep(1);
                fprintf(fp, "Logging info...\n");
                fflush(fp);
                // Implement and call some function that does core work for this daemon.
        }
        fclose(fp);
        return (0);
}
```