Προγραμματισμός συστημάτων UNIX/POSIX

Διεργασίες (processes)

MYY502

Δομή αρχείου προγράμματος

- **Πρόγραμμα** (program) ονομάζεται το εκτελέσιμο αρχείο που βρίσκεται αποθηκευμένο στο δίσκο (π.χ. το “a.out”, ή το “ls” ή το “gcc”)
  - Προγράμματα φτιάχνουμε εμείς (a.out) ή
  - Είναι έτοιμες εφαρμογές (/usr/bin/gcc, /bin/ls, …)
- Τα προγράμματα είναι συνηθισμένα δυαδικά **αρχεία**. Απλά έχουν την ιδιότητα ότι μπορούν να **εκτελεστούν** (δηλαδή έχουν εντολές μηχανής).

- Τι περιέχει το αρχείο a.out? Ο μεταφραστής όταν δημιουργεί το πρόγραμμα, αποθηκεύει μέσα στο a.out διάφορα **τμήματα (segments):**
  1. Τις **εντολές** του προγράμματος (σε γλώσσα μηχανής) – *ονομάζεται **text ή code segment***
  2. Πληροφορίες για τις **αρχικοποιημένες καθολικές μεταβλητές και σταθερές** που έχει το πρόγραμμα (μεγέθη, αρχικές τιμές) – *ονομάζεται **data segment***
  3. Πληροφορίες για τις **μη αρχικοποιημένες καθολικές μεταβλητές** που έχει το πρόγραμμα (μεγέθη) – *ονομάζεται **bss segment***
- Μπορείτε να δείτε λεπτομέρειες των τμημάτων με την εντολή **`size`**, π.χ. εκτελέστε στο τερματικό:

```
$ size a.out
```

Παράδειγμα

```
int a = 5, arr1[50] = { 1, 2, 3 };
int b, arr2[50];

int main() {
  char *name = "Hi!";
  int  c;
  c = a + b + strlen(name);
  return 0;
}
```

Αρχικοποιημένες καθολικές μεταβλητές και σταθερά strings βρίσκονται στο τμήμα data του a.out

Μη αρχικοποιημένες καθολικές μεταβλητές βρίσκονται στο τμήμα bss του a.out

Οι τοπικές μεταβλητές δεν βρίσκονται πουθενά! **Θα** τοποθετηθούν στη στοίβα όταν το πρόγραμμα εκτελείται ως διεργασία.

Οι εντολές βρίσκονται στο τμήμα text του a.out

Πρόγραμμα (program) & διεργασία (process)

- **Πρόγραμμα** (program) ονομάζεται το εκτελέσιμο αρχείο που βρίσκεται αποθηκευμένο στο δίσκο (π.χ. το “a.out”, ή το “ls” ή το “gcc”)
  - Προγράμματα φτιάχνουμε εμείς (a.out) ή
  - Είναι έτοιμες εφαρμογές (/usr/bin/gcc, /bin/ls, …)
- Όταν ένα πρόγραμμα εκτελείται γίνεται **διεργασία** (process).
- Ανάλογα με το σύστημα, μπορεί να συνυπάρχουν και να εκτελούνται **πολλές** διεργασίες σε κάθε χρονική στιγμή (multitasking)
  - Π.χ. έχω ανοιχτό τον browser, τον editor και το τερματικό και την ίδια στιγμή ακούω κι ένα τραγούδι.
  - Μοιράζονται τον επεξεργαστή (ή τους επεξεργαστές), τη μνήμη, κλπ.
  - Αν υπάρχουν πολλοί επεξεργαστές (παράλληλοι υπολογιστές), μπορεί να εκτελούνται *ταυτόχρονα* αλλιώς μπορεί να εναλλάσσονται με πολύ γρήγορο ρυθμό και να εκτελούνται *εκ περιτροπής* δίνοντας την *ψευδαίσθηση* της ταυτόχρονης εκτέλεσης (timesharing).

Διεργασίες

- Η διαδικασία κατά την οποία το πρόγραμμα μεταλλάσσεται σε διεργασία είναι ιδιαίτερα πολύπλοκη και την αναλαμβάνει το λειτουργικό σύστημα του υπολογιστή.
- «Χοντρικά» βήματα:
  1. Ανοίγεται το αρχείο του προγράμματος από τον δίσκο.
  2. Μεταφέρονται (αντιγράφονται) οι *εντολές* του (από το τμήμα text) σε κάποιο σημείο στην κύρια μνήμη
  3. Δεσμεύεται χώρος για την τοποθέτηση των *καθολικών* μεταβλητών (πληροφορίες από τα τμήματα data και bss)
  4. Αρχικοποιούνται οι καθολικές μεταβλητές (από το τμήμα data)
  5. Δεσμεύεται χώρος μνήμης για τη στοίβα του προγράμματος (stack)
  6. Προετοιμάζεται ο χώρος μνήμης από τον οποίο θα δίνονται bytes αν το πρόγραμμα καλεί την malloc (ο χώρος αυτός ονομάζεται *σωρός* – *heap*)
  7. Προετοιμάζονται πολύπλοκες δομές του λειτουργικού συστήματος για διαχείριση και δρομολόγηση της διεργασίας
  8. Εν τέλει, καλείται η `main()`.

Μερικά για τις διεργασίες

- Μπορεί πολλές διεργασίες να προέρχονται από το **ίδιο** πρόγραμμα
  - Π.χ. έχω ανοιχτά πολλά τερματικά: ένα είναι το πρόγραμμα του τερματικού στον δίσκο, όμως εκτελούνται πολλές *αυτόνομες και ξεχωριστές διεργασίες.*
- Κάθε διεργασία παίρνει μία μοναδική **ταυτότητα (process id)** – είναι ένας ακέραιος αριθμός που της δίνει το λειτουργικό σύστημα
  - Η ταυτότητα είναι εν γένει τυχαία. Κάθε φορά που εκτελούμε το ίδιο πρόγραμμα, το σύστημα μπορεί να δίνει στην αντίστοιχη διεργασία διαφορετική ταυτότητα.
  - Διεργασίες που εκτελούνται ταυτόχρονα και οι οποίες προέρχονται από το ίδιο πρόγραμμα θεωρούνται εντελώς διαφορετικές, ανεξάρτητες και έχουν διαφορετικές ταυτότητες.
- Μπορώ να βρω ποιες διεργασίες εκτελούνται αυτή τη στιγμή, μαζί με τις ταυτότητές τους, εκτελώντας:

```
$ ps
```

Πώς μπορώ να βρω την ταυτότητα της διεργασίας

- 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
```

Η κλήση exec

- Μέσα από το πρόγραμμά μου μπορώ να εκτελέσω ένα οποιοδήποτε άλλο πρόγραμμα με την χρήση των κλήσεων **exec**.
  - Η διεργασία μου θα **αντικατασταθεί** και θα εκτελέσει το νέο πρόγραμμα.
  - Δηλαδή «εξαφανίζεται» ο κώδικας και τα δεδομένα της διεργασίας μου και στη θέση τους μπαίνουν ο κώδικας και τα δεδομένα του νέου προγράμματος που θέλω να εκτελέσω.
  - Το process id προφανώς δεν αλλάζει μιας και δεν φτιάχτηκε άλλη διεργασία. **Άλλαξε το πρόγραμμα αλλά όχι η διεργασία!!**
    - Επομένως, παρέμειναν πολλά πράγματα ΙΔΙΑ, όπως π.χ το user id, ο τρέχων φάκελος εργασίας, *τα αρχεία που ήταν ανοιχτά* κλπ
- Υπάρχουν αρκετές εκδόσεις της exec
  - execl(), execv(), execlp(), execvp(), execle(), execve()
  - Κάνουν χοντρικά την ίδια δουλειά, αλλά με άλλα ορίσματα / επιλογές.

Η κλήση execl()

- Απαιτείται `#include <unistd.h>`
- Η `execl()` παίρνει ως ορίσματα **την πλήρη διαδρομή του προγράμματος** (αρχείου) που θέλω να εκτελέσω και στη συνέχεια οι επόμενες παράμετροι είναι όλα τα ορίσματα που πρέπει να περαστούν στην main() του προγράμματος που θα εκτελεστεί (με πρώτο το όνομα του προγράμματος, όχι την πλήρη διαδρομή του).
  - Η `execl()` προφανώς είναι variadic.
- Η τελευταία παράμετρος πρέπει να είναι `NULL`.
- Αν όλα πάνε καλά, η `execl()` δεν επιστρέφει ποτέ (έχει καταστραφεί η διεργασία που την κάλεσε…)
- Παράδειγμα: για να εκτελέσω “gcc –c myfile.c” θα πρέπει να καλέσω την `execl()` κάπως έτσι:

```
execl("/usr/bin/gcc", "gcc", "-c", "myfile.c", NULL);
      ΠΛΗΡΗΣ ΔΙΑΔΡΟΜΗ   Όνομα   Όρισμα    Όρισμα     Τέλος
```

Παράδειγμα execl 1/3

```
#include <stdio.h>
#include <unistd.h>  /* Needed for execl() */

int main() {
  printf("Before calling execl\n");
  /* Εκτέλεση "ls –l"; Δηλαδή "ls" με 1 όρισμα, το "-l" */
  execl("ls", "ls", "-l", NULL); /* NULL μετά το τελευταίο όρισμα */
  printf("After calling execl\n");
  return 0;
}
```

- Εκτέλεση στο τερματικό:

```
$ ./a.out
Before calling execl
After calling execl
$
```

- Τι έγινε???

Το “ls” δεν είναι η ΠΛΗΡΗΣ διαδρομή της εντολής ls! Πρέπει να περάσουμε **ολόκληρο το μονοπάτι** όπου βρίσκεται αποθηκευμένο το πρόγραμμα ls.

Πώς το βρίσκουμε;

```
$ which ls
/bin/ls
$
```

Παράδειγμα execl 2/3

```
#include <stdio.h>
#include <unistd.h>  /* Needed for execl() */

int main() {
  printf("Before calling execl\n");
  /* Εκτέλεση "ls –l"; Δηλαδή "ls" με 1 όρισμα, το "-l" */
  execl("/bin/ls", "ls", "-l", NULL);
  printf("After calling execl\n");
  return 0;
}
```

- Εκτέλεση στο τερματικό:

```
$ ./a.out
Before calling execl
total 7
-rwxr-xr-x 1 dimako staff 4688 May 14 22:23 a.out
-rw-r--r-- 1 dimako staff  167 May 14 22:23 test_exec.c
$
```

- Σωστή εκτέλεση! Το δεύτερο `printf()` δεν εκτελέστηκε ποτέ…

Παράδειγμα execl 3/3

- Το παρακάτω είναι ο κώδικας του προγράμματος a.out. Τι θα γίνει όταν το εκτελέσουμε;

```
#include <stdio.h>
#include <unistd.h>  /* Needed for execl() */

int main() {
  printf("Before calling execl, pid = %d\n", getpid());
  execl("./a.out", "a.out", NULL);
  printf("After calling execl\n");
  return 0;
}
```

- Θα εκτελείται επ' άπειρο το a.out από την ίδια διεργασία και άρα:

```
$ ./a.out
Before calling execl 11454
Before calling execl 11454
Before calling execl 11454
Before calling execl 11454
Before calling execl 11454
…
```

Η κλήση execv()

- Πρόκειται για ίδια λογική με την `execl()`, μόνο που τα ορίσματα δεν δίνονται ξεχωριστά, αλλά μαζεμένα όλα μαζί σε έναν πίνακα:

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

- Το προηγούμενο παράδειγμα με χρήση της `execv()`:

```
#include <stdio.h>
#include <unistd.h>  /* Needed for execl() */

int main() {
   /* Εκτέλεση "ls -l"; Πίνακας με 3 δείκτες σε συμβολοσειρές */
  char *arg[3] = {"ls", "-l", NULL };

  printf("Before calling execv\n");
  execv("/bin/ls", arg);
  return 0;
}
```

Προγραμματισμός συστημάτων UNIX/POSIX

Δημιουργία νέων διεργασιών (process creation – fork)

Δημιουργία νέας διεργασίας

- Μπορώ μέσα από το πρόγραμμά μου μπορώ να δημιουργήσω μία εντελώς νέα διεργασία;
  - 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 (περίεργη) κλήση συστήματος `fork()`:
  - Η συνάρτηση αυτή δημιουργεί ένα **πανομοιότυπο αντίγραφο** της τρέχουσας διεργασίας και οι δύο, πλέον, διεργασίες εκτελούνται ταυτόχρονα και ανεξάρτητα!
  - Η διεργασία που κάλεσε την `fork()` ονομάζεται **διεργασία-γονέας (parent)** ενώ η νέα διεργασία ονομάζεται **θυγατρική ή διεργασία-παιδί (child process)**.

fork

- 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.
- Παράδειγμα:

```
#include <unistd.h>  /* Needed for fork(), getpid() */

int main() {
  printf("parent (%d) about to call fork\n", getpid());
  fork();
  printf("Hi from process %d\n", getpid());
  return 0;
}
```

- Εκτέλεση στο τερματικό:

```
$ ./a.out
parent (11533) about to call fork
Hi from process 11533
Hi from process 11534
$
```

Εξήγηση

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

int main() {
  printf("parent (%d) about to call fork\n",
         getpid());
  fork();
  printf("Hi from process %d\n", getpid());
  return 0;
}
```

Η γονική διεργασία εκτελεί την `printf`

Η γονική διεργασία καλεί την `fork`

1) Δημιουργείται νέα διεργασία (θυγατρική). Πρόκειται για διεργασία-αντίγραφο του γονέα, η οποία εκτελεί το ίδιο πρόγραμμα.
2) Και η γονική και η θυγατρική διεργασία συνεχίζουν από την κλήση της `fork` και κάτω.

Αυτό το `printf` το εκτελούν και οι δύο διεργασίες (οι οποίες είναι διαφορετικές και άρα έχουν διαφορετικό process id)

```
$ ./a.out
parent (11533) about to call fork
Hi from process 11533
Hi from process 11534
$
```

Ερωτήματα & παρατηρήσεις

- Γιατί τυπώθηκε πρώτα το process id του γονέα και μετά του παιδιού; Προλαβαίνει πάντα και εκτελείται πρώτα ο γονέας;
  - Απάντηση: Είναι εντελώς τυχαίο! Αν ξαναεκτελέσουμε το a.out μπορεί να εκτυπωθεί πρώτα το μήνυμα του παιδιού…
- Δεν είναι μάλλον ανούσιο το γεγονός ότι γονέας και παιδί είναι πανομοιότυποι και κάνουν ακριβώς τα ίδια πράγματα;
  - Απάντηση: Προφανώς ναι! Μπορούμε όμως να τους βάλουμε να κάνουν διαφορετικά πράγματα.
- Πώς μπορεί το παιδί να γνωρίζει ότι είναι παιδί και να κάνει άλλα πράγματα από αυτά που κάνει ο γονέας;
  - Απάντηση: Πρέπει να κοιτάξουμε την τιμή που επιστρέφει η fork(). Αν είναι ίση με 0, τότε βρισκόμαστε στο παιδί, αλλιώς στον γονέα!
  - Στον γονέα, επιστρέφει το process id του παιδιού.

Διαφοροποίηση γονέα – παιδιού

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

int main() {
  int ret;
  printf("parent (%d) about to call fork\n", getpid());
  ret = fork();   /* Remember what fork() returned */
  if (ret == 0)   /* 0 returned to child */
    printf("I am the child, with pid %d\n", getpid());
  else            /* child's pid (> 0) returned to parent */
    printf("I am the parent, with pid %d; child pid = %d\n",
           getpid(), ret);
  return 0;
}
```

- Εκτέλεση στο τερματικό:

```
$ ./a.out
parent (11644) about to call fork
I am the parent, with pid 11644; child pid = 11645
I am the child, with pid 11645
$
```

getppid() – η ταυτότητα του γονέα

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

int main() {
  int ret;
  printf("parent (%d) about to call fork\n", getpid());
  ret = fork();    /* Remember what fork() returned */
  if (ret == 0)    /* 0 returned to child */
    printf("I am the child, with pid %d; parent pid = %d\n",
           getpid(), getppid());
  else             /* child's pid (> 0) returned to parent */
    printf("I am the parent, with pid %d; child pid = %d\n",
           getpid(), ret);
  return 0;
}
```

- Εκτέλεση στο τερματικό:

```
$ ./a.out
parent (11640) about to call fork
I am the parent, with pid 11640; child pid = 11645
I am the child, with pid 11645; parent pid = 11640
$
```

Το παιδί είναι αντίγραφο του πατέρα

- Η θυγατρική διεργασία είναι πλήρες και πανομοιότυπο αντίγραφο της γονικής διεργασίας
- Τρέχει αντίγραφο του ίδιου προγράμματος με τον γονέα
- Ακόμα πιο σημαντικό: έχει πανομοιότυπα αντίγραφα όλων των μεταβλητών του γονέα, **με τις τιμές που είχαν στον πατέρα πριν τη fork()**.

- Γενικότερα, *κληρονομεί τα πάντα από τον πατέρα*. Ακόμα και όσα αρχεία είχε κάνει open() ο πατέρας, τα έχει και το παιδί και μάλιστα είναι ήδη ανοικτά!

Όλα ίδια – παράδειγμα

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

int a = 5;

int main() {
  int b = 10;
  printf("parent about to call fork\n");
  if (fork() == 0) /* 0 returned to child */
    printf("Child (pid %d): a = %d, b = %d\n", getpid(), a, b);
  else
    printf("Parent (pid %d): a = %d, b = %d\n", getpid(), a, b);
  return 0;
}
```

- Εκτέλεση στο τερματικό:

```
$ ./a.out
parent about to call fork
Child (pid 11704): a = 5, b = 10
Parent (pid 11703): a = 5, b = 10
$
```

Το παιδί είναι διαφορετική διεργασία

- Μπορεί να δημιουργείται ως πλήρες αντίγραφο του γονέα, όμως στη συνέχεια είναι αυτόνομο και
  a) Μπορεί να εκτελεί άλλες εντολές του προγράμματος από ότι ο γονέας
  b) Μπορεί να δίνει άλλες τιμές στις μεταβλητές, μιας και είναι δικές του, διαφορετικές από του γονέα.
- Ότι αλλαγές κάνει στις μεταβλητές του το παιδί, **δεν** επηρεάζουν τις αντίστοιχες μεταβλητές του πατέρα (και το αντίστροφο)

Διαφορετικές τιμές μεταβλητών – παράδειγμα

```
#include <stdio.h>
#include <unistd.h>
int a = 5;

int main() {
  int b = 10;
  if (fork() == 0) { /* 0 returned to child */
    printf("child: a = %d, b = %d\n", a, b);
    a++; b++;
    printf("child now: a = %d, b = %d\n", a, b);
  }
  else {
    printf("parent: a = %d, b = %d\n", a, b);
    a--; b--;
    printf("parent now: a = %d, b = %d\n", a, b);
  }
  return 0;
}
```

- Εκτέλεση στο τερματικό:

```
$ ./a.out
parent: a = 5, b = 10
child: a = 5, b = 10
parent now: a = 4, b = 9
child now: a = 6, b = 11
$
```

Περιμένοντας το παιδί – waitpid()

- Όταν δημιουργείται μία νέα διεργασία, αυτή δρομολογείται και εκτελείται ανεξάρτητα από τον γονέα.
  - Μπορεί να τερματίσει ο γονέας και το παιδί να συνεχίσει να εκτελείται ή και το ανάποδο.
- Στις περισσότερες περιπτώσεις όμως, όταν φτιάξουμε μία νέα διεργασία, μας ενδιαφέρει να την περιμένουμε να ολοκληρώσει τη δουλειά της ή τέλος πάντων να γνωρίζουμε με κάποιο τρόπο ότι τερματίστηκε.
- Για αυτό το σκοπό χρησιμοποιείται η κλήση `waitpid()` η οποία μας επιτρέπει να σταματήσουμε και να περιμένουμε να τερματιστεί μία συγκεκριμένη διεργασία-παιδί

waitpid()

- Απαιτείται `#include <sys/wait.h>`

  `pid_t waitpid(pid_t pid, int *exitstatus, int options);`
- Ο τύπος `pid_t` είναι κατά βάση ψευδώνυμο του `int`.
- Το πρώτο όρισμα (`pid`) είναι το process id της θυγατρικής διεργασίας που περιμένουμε να τελειώσει.
  - Αν περάσουμε **-1**, τότε περιμένουμε **μία οποιαδήποτε από τις θυγατρικές μας διεργασίες** να τελειώσει.
- Το τρίτο όρισμα δεν μας ενδιαφέρει – *θα δίνουμε πάντα 0*.
- Το δεύτερο όρισμα δείχνει σε έναν ακέραιο, στον οποίο θα αποθηκευτεί **η κατάσταση τερματισμού** (βασικά περιέχει την τιμή επιστροφής) της θυγατρικής διεργασίας.
  - Με αυτό τον τρόπο, το παιδί μπορεί επίσης να ενημερώσει τον γονέα για το τι έγινε.
  - Αν δεν μας ενδιαφέρει η τιμή, μπορούμε να περάσουμε NULL.

wait()

- Η κλήση

```
wait(&exitstatus);
```

- Είναι ισοδύναμη με την κλήση:

```
waitpid(-1, &exitstatus, 0);
```

- Δηλαδή, περιμένει τον τερματισμό οποιουδήποτε παιδιού.
- Την αγνοούμε επομένως…

Παράδειγμα αναμονής 1

```
#include <stdio.h>
#include <stdlib.h>     /* For exit() */
#include <unistd.h>     /* For fork(), getpid() */
#include <sys/wait.h>   /* For waitpid() */

int main() {
  int pid;

  pid = fork();
  if (pid == 0) {  /* child */
    return 5;      /* child exits with 5 */
  }
  printf("parent (%d) waits for child (%d)...\n", getpid(), pid);
  waitpid(pid, NULL, 0);
  printf("child terminated!\n");
  return 0;
}
```

- Η θυγατρική διεργασία τερματίζει κανονικά με `return` από την `main()` της

Παράδειγμα αναμονής 2

```
#include <stdio.h>
#include <stdlib.h>     /* For exit() */
#include <unistd.h>     /* For fork(), getpid() */
#include <sys/wait.h>   /* For waitpid() */

void delay() {      /* Just delay */
  int i, sum=0;
  for (i = 0; i < 10000000; i++)
    sum += i;
  printf("child (%d) exits...\n", getpid());
  exit(5);    /* Child exits with 5 */
}

int main() {
  int pid;

  pid = fork();
  if (pid == 0)   /* child */
    delay();
  printf("parent (%d) waits for child (%d)...\n", getpid(), pid);
  waitpid(pid, NULL, 0);
  printf("child terminated!\n");
  return 0;
}
```

- Η θυγατρική διεργασία τερματίζει κανονικά με `exit()`

Παίρνοντας την τιμή τερματισμού

- Για να μπορέσω να βρω με τι τιμή τερμάτισε το παιδί μου (είτε με `exit` είτε με `return`), θα πρέπει να περάσω τη δεύτερη παράμετρο στην `waitpid()` με δείκτη σε κάποιον ακέραιο
- Στον ακέραιο αυτό, κατά την επιστροφή από την κλήση θα περιλαμβάνονται 2 ειδών πληροφορίες
  - α) Αν τερματίστηκε κανονικά (με return ή exit) η διεργασία-παιδί ή διακόπηκε απρόσμενα π.χ. λόγω κάποιου σφάλματος ή ctrl-C κλπ.
  - β) Η τιμή τερματισμού αν τερματίστηκε κανονικά
- Για να διαπιστώσω το α) πρέπει να χρησιμοποιήσω το macro `WIFEXITED()`.
- Αν αυτό μου δώσει TRUE, τότε το β) προέκυπτε από το macro `WEXITSTATUS()`.

Παράδειγμα αναμονής 3

```
#include <stdio.h>
#include <stdlib.h>     /* For exit() */
#include <unistd.h>     /* For fork(), getpid() */
#include <sys/wait.h>   /* For waitpid() */

void delay() {      /* Just delay */
  int i, sum=0;
  for (i = 0; i < 10000000; i++)
    sum += i;
  printf("child (%d) exits...\n", getpid());
  exit(5);    /* Child exits with 5 */
}

int main() {
  int pid, status;

  pid = fork();
  if (pid == 0)   /* child */
    delay();
  printf("parent (%d) waits for child (%d)...\n", getpid(), pid);
  waitpid(pid, &status, 0);
  if (WIFEXITED(status)) /* Terminated OK? */
    printf("child exited normally with value %d\n", WEXITSTATUS(status));
  else
    printf("child was terminated abnormaly.\n");
  return 0;
}
```

Λεπτομέρειες – Παρατηρήσεις – waitid()

- Το macro `WEXITSTATUS()` *επιστρέφει **ΜΟΝΟ** τα 8 λιγότερο σημαντικά bits της τιμής τερματισμού του παιδιού*
  - Επομένως, αν το παιδί θέλει να «πει» κάτι στον γονέα μέσω exit/waitpid, αυτό θα πρέπει να είναι ένας αριθμός μέχρι το 255.

- Η πιο σύγχρονη εκδοχή της `waitpid()` είναι η συνάρτηση `waitid()`:

```
waitid(P_PID, childpid, &info, WEXITED);
```

Για αναμονή *οποιουδήποτε* παιδιού:

```
waitid(P_ALL, 0, &info, WEXITED);
```

όπου το `info` είναι τύπου `siginfo_t`.

  - Στο `info.si_status` μπορεί κανείς να βρει την τιμή τερματισμού του παιδιού, και μάλιστα ***ΟΛΟΚΛΗΡΟ*** *τον ακέραιο* (όχι μόνο τα 8 λιγότερο σημαντικά bits).

Από το τερματικό, για να βρούμε λεπτομέρειες και πληροφορίες για συναρτήσεις, χρησιμοποιούμε την εντολή **man:**

```
$ man waitid
```

Fork + exec

- Μία πολύ συνηθισμένη περίπτωση στην πράξη είναι να θέλουμε να δημιουργήσουμε μία θυγατρική διεργασία η οποία θα εκτελέσει ένα άλλο πρόγραμμα και θα τερματίσει.
- Αυτό το σενάριο υλοποιείται με τη χρήση της `fork()` για να δημιουργηθεί η νέα διεργασία και στη συνέχεια με το παιδί να καλεί την `exec()` για να εκτελέσει άλλο πρόγραμμα
- Ο γονέας, ανάλογα με την περίπτωση, μπορεί να περιμένει ή όχι τον τερματισμό του παιδιού.

Παράδειγμα fork + exec

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

void childfunc() {     /* Just delay */
  printf("child (%d) will fire up a firefox window!\n", getpid());
  execl("/usr/bin/firefox", "firefox", NULL);
  exit(5);
}

int main() {
  printf("about to fork...\n");
  if (fork() == 0) /* child */
    childfunc();
  printf("parent (%d) waits for child...\n", getpid());
  waitpid(-1, NULL, 0);   /* or just wait(NULL) */
  return 0;
}
```

Η συνάρτηση system()

- Πρόκειται για μία συνάρτηση η οποία κάνει τη λειτουργία των fork + exec + waitpid με μία μόνο κλήση!
  - Δημιουργεί ένα παιδί, το οποίο εκτελεί το πρόγραμμα που θέλουμε (μέσω shell) και ο γονέας επιστρέφει μόλις τερματίσει το παιδί.
- Η συνάρτηση `system()`:
  - Η `system()` παίρνει ως όρισμα 1 συμβολοσειρά, η οποία περιέχει ΑΚΡΙΒΩΣ την εντολή που θα εκτελούσαμε στο τερματικό!
    a) Μπλοκάρει (σταματά προσωρινά) την τρέχουσα διεργασία
    b) δημιουργεί μία νέα διεργασία η οποία εκτελεί ένα κέλυφος (shell, π.χ. Bash)
    c) το κέλυφος εκτελεί την εντολή που περάσαμε ως παράμετρο
    d) τερματίζει η διεργασία του κελύφους και
    e) συνεχίζει η αρχική μας διεργασία.
  - Απλός, γρήγορος αλλά και λειτουργικά περιοριστικός τρόπος.

Παράδειγμα με system()

- Παράδειγμα:

```
#include <stdio.h>
#include <stdlib.h> /* For system() */

int main() {
  printf("Before system()\n");
  system("ls -l");  /* Create shell to execute ls */
  printf("After system()\n");
  return 0;
}
```

- Εκτέλεση στο τερματικό:

```
$ ./a.out
Before system()
total 16
-rwxr-xr-x   1 dimako   staff   6748 May 14 14:49 a.out
-rw-r--r--   1 dimako   staff    236 May 14 14:49 test.c
After system()
```