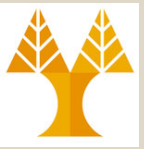
ΕΠΛ421 - Προγραμματισμός Συστημάτων

Διάλεξη 13: Διεργασίες: Περιβάλλον και Έλεγχος (Processes: Environment & Control)

(Κεφάλαια 7,8 - Stevens & Rago)

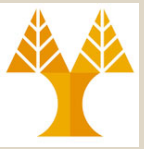
Δημήτρης Ζεϊναλιπούρ

Περιεχόμενο Διάλεξης

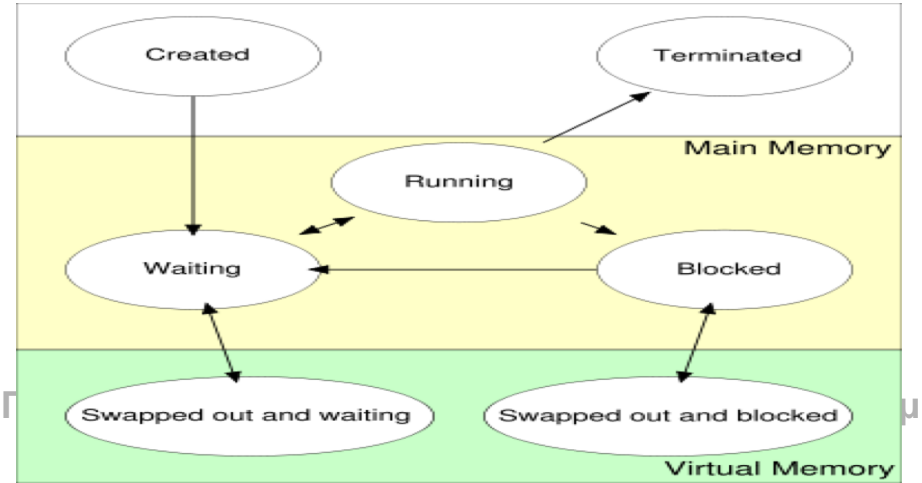


Α. **Διεργασίες**: **Εισαγωγικές Έννοιες**. (7.1,8.1)
Β. **Ταυτότητες** Διεργασιών (getpid(), getppid(), getuid(), getgid()) (8.2)
Γ. **Διεργασίες στην Μνήμη** (Δομή & Αναπαράσταση) (7.5-7.6)
Δ. **Δημιουργία** Διεργασιών (fork()) (8.3)
Ε. **Μεταβλητές** Περιβάλλοντος (7.9)
Ζ. **Ορφανές** Διεργασίες (8.3)
Η. Διεργασίες και **Αρχεία** (8.3)
Θ. **Αναμονή Διεργασιών** (wait(), waitpid()) (8.6)
Ι. **Zombie** Διεργασίες (8.5)

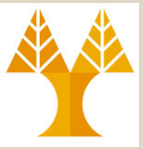
Α. Διεργασίες - Εισαγωγή



- **Διεργασία:** *Ένα πρόγραμμα υπό εκτέλεση*
- Κάθε Διεργασία έχει μια μοναδική ταυτότητα "process id".
- Εάν και μοναδική, αυτή η ταυτότητα **επανά-χρησιμοποιείται** από το σύστημα όταν τερματιστεί η εκτέλεση μιας διεργασίας (και μετά από κάποιο χρόνο)
- Μια διεργασία μπορεί να βρίσκεται σε διάφορες **καταστάσεις (states)** όπως φαίνεται στο πιο κάτω **διάγραμμα καταστάσεων** (**state diagram**).



Β. Ταυτότητες Διεργασιών



Εκτέλεση από το κέλυφος: $ps -ef | head -15 (στο linux)

Userid, processid, parentprocessid, cpu%, systemtime, command

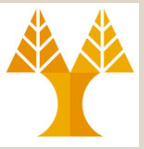
UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	0	Feb15	?	00:00:24	init
root	2	1	0	Feb15	?	00:00:00	[keventd]
root	3	1	0	Feb15	?	00:00:26	[ksoftirqd_CPU0]
root	4	1	0	Feb15	?	00:00:00	[ksoftirqd_CPU1]
root	5	1	0	Feb15	?	00:00:00	[ksoftirqd_CPU2]
root	6	1	0	Feb15	?	00:00:00	[ksoftirqd_CPU3]
root	7	1	0	Feb15	?	00:12:49	[kswapd]
root	8	1	4	Feb15	?	13:10:36	[kscand]
root	9	1	0	Feb15	?	00:00:00	[bdflush]
root	10	1	0	Feb15	?	00:00:32	[kupdated]
			0	Feb15	?	00:00:00	[mdrecoveryd]
			0	Feb15	?	00:00:00	[scsi_eh_0]
			0	Feb15	?	00:00:00	[scsi_eh_1]
			0	Feb15	?	00:02:59	[kjournald]

a) Η init γεννιέται από τον χρονοδρομολογητή του πυρήνα PID#0

b) Όλες οι διεργασίες «γεννιούνται» από την Init η οποια εχει PID#1

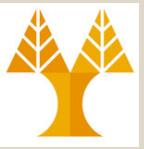
c) Οι διεργασίες αυτές εκτελούνται στο background και ονομάζονται **daemon processes.** Για αυτό τον λόγο δεν έχουν controlling terminal (stdin,out,err).

Β. Ταυτότητες Διεργασιών



- Μερικές ταυτότητες (συνήθως με μικρές τιμές) είναι κρατημένες για διεργασίες του συστήματος, π.χ.,
  - **PID#0 : Ο Χρονοδρομολογητής (scheduler ή swapper ή dispatcher)** του πυρήνα.
    - Αποφασίζει ποιες διεργασίες θα εκτελεστούν (run) και ποιες θα περιμένουν (wait).
  - **PID#1 : Ο Εκκινητής Υπηρεσιών (/sbin/init).**
    - Αυτή η διεργασία "γεννιέται" από τον πυρήνα κατά την διάρκεια της εκκίνησης (boot).
    - Η **init** αναλαμβάνει να **ξεκινήσει** τις διάφορες υπηρεσίες του UNIX (όπως αυτές ορίζονται στα **/etc/rc*** και το **/etc/inittab**).
    - Θα δούμε ότι το init **γίνεται πατέρας** κάθε διεργασίας που χάνει τον γονέα της.
  - **PID#7 : Η Διεργασία Σελιδοποίησης (Page daemon) (pager)** του πυρήνα
    - Υποστηρίζει την λειτουργία της νοητής μνήμης (virtual memory)
  - **….**

Β. Ταυτότητες Διεργασιών



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

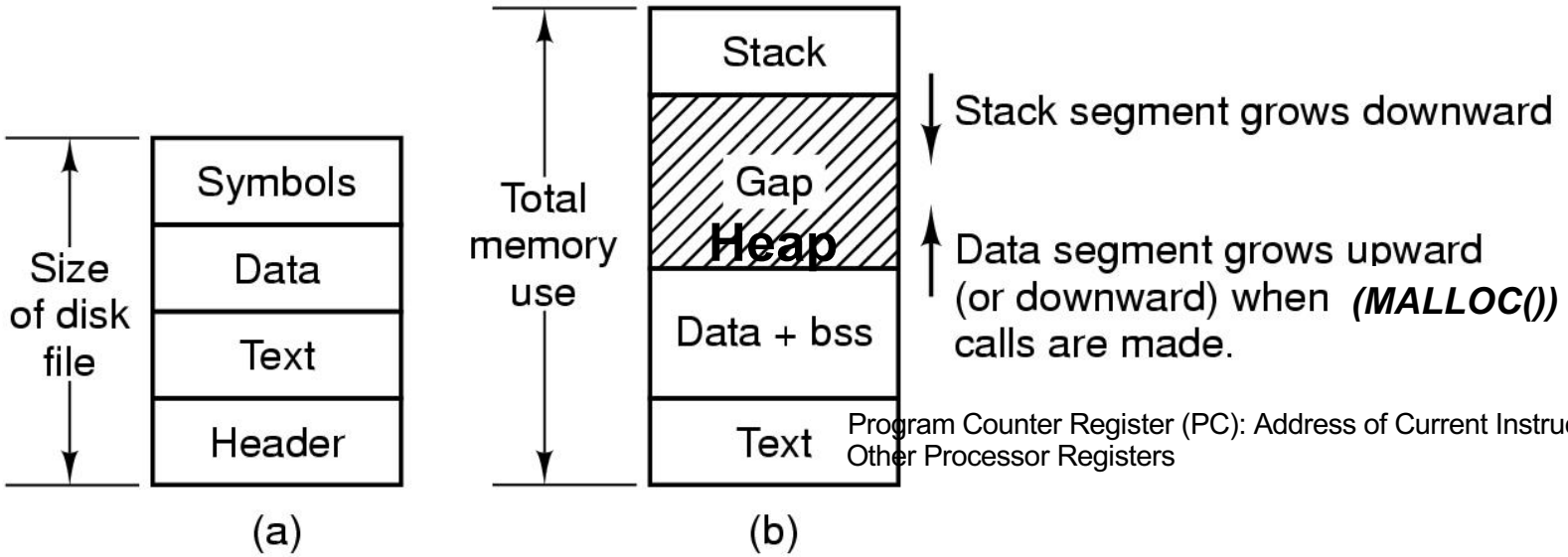
```
#include <unistd.h>
pid_t getpid(void);   // ProcessID διεργασίας
pid_t getppid(void); // ProcessID πατέρα
pid_t getuid(void);   // Real UserID διεργασίας (βλέπε /etc/passwd)
pid_t getgid(void);   // Real GroupID διεργασίας (βλέπε /etc/passwd)
```

```
#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.

Γ. Διεργασίες στην Μνήμη

- Προτού δούμε περισσότερα για τις διεργασίες ας θυμηθούμε πως αναπαρίσταται μια διεργασία στη μνήμη.
- Αρχικά το εκτελέσιμο αρχείο (στην δευτερεύουσα μνήμη) έχει μια συγκεκριμένη δομή (αριστερό σχήμα) όπως αυτή διαμορφώνεται μετά την μεταγλώττιση.
- Όταν φορτωθεί στην μνήμη το πρόγραμμα τότε έχει την δομή που φαίνεται στα δεξιά.



Program Counter Register (PC): Address of Current Instruction
Other Processor Registers

Δ. Δημιουργία Διεργασιών

Πότε χρειάζεται;

- Εάν μια διεργασία θέλει να **κλωνοποιήσει** τον εαυτό της, έτσι ώστε να εκτελέσει παράλληλα διαφορετικό μέρος του κώδικα.
- Αυτό είναι ιδιαίτερα χρήσιμο με **multi-processors** (συστήματα πολυεπεξεργαστών) αλλά και **single-processors** (συστήματα ενός επεξεργαστή)
  - Π.χ. Ένας Webserver Α είναι μια διεργασία που έχει ανοικτό ένα socket στην θύρα 80 (τύπος αρχείου).
  - Όταν λάβει ο Α μια αίτηση μέσω του socket από κάποιο web-browser, ο Α κάνει fork() μια νέα διεργασία. Με αυτό τον τρόπο μπορούν πολλοί χρήστες να εξυπηρετούνται ταυτόχρονα από τον Α.
  - Θα δούμε στην συνέχεια ότι ο webserver υλοποιείται ακόμη καλύτερα με νήματα (threads)…

Δ. Δημιουργία Διεργασιών
Η κλήση συστήματος fork()

- Η συνάρτηση **fork() (διακλάδωση)** είναι ο μόνος τρόπος δημιουργίας μιας νέας διεργασίας στο περιβάλλον UNIX.
- Η διεργασία Α γονέας δημιουργεί μια διεργασία παιδί Β.
- Το Β είναι **πιστό αντίγραφο** του Α (με μερικές μικρές διαφορές τις οποίες θα δούμε στην συνέχεια).

```
#include <unistd.h>
pid_t fork(void);  // άτυπα το pid_t είναι int
```

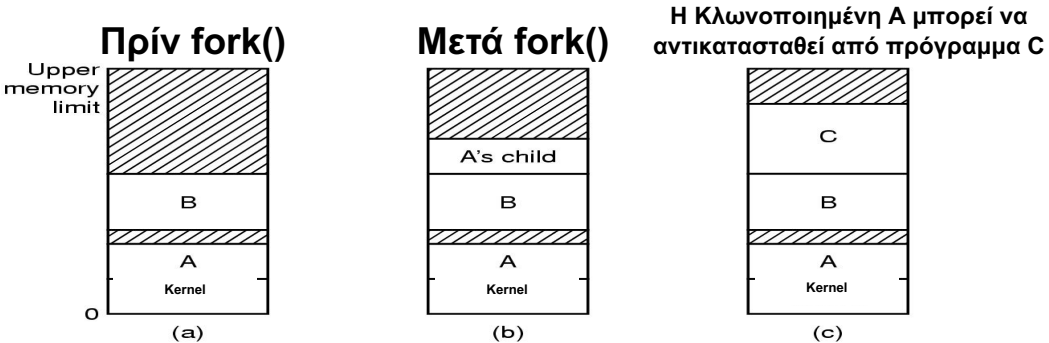
- **Επιστρέφει 0 στο παιδί**,
- Επιστρέφει το processID του παιδιού στο γονέα ή -1 σε περίπ. λάθους.

- Η συνάρτηση fork() **εκτελείται** μια **φορά** αλλά **επιστρέφει δυο τιμές** !!! …. μια επιστροφή γίνεται στο **γονέα** και μια επιστροφή στο **παιδί**.
- Μια διεργασία-παιδί μπορεί κατά την διάρκεια της «δημιουργίας» να **διαφοροποιήσει τον εαυτό** της από τον γονέα της … θα μελετηθεί στην συνέχεια.
  - Ουσιαστικά, μια διεργασία μπορεί να **αντικαταστήσει** με την **exec()** τον κώδικα, τα δεδομένα, και την στοίβα της με αυτά ενός **εκτελέσιμου αρχείου**

Δ. Δημιουργία Διεργασιών
Ο Μηχανισμός της fork()

- Σε ένα πρόγραμμα A καλούμε κάπου την **εντολή fork().**
- Αυτό δημιουργεί ένα **πιστό αντίγραφο** της διεργασίας A (A's child), δηλαδή αντιγράφονται κάπου αλλού στην μνήμη τα **δεδομένα (data),** η **στοίβα (program stack),** η **σωρός (heap),… οτιδήποτε βρίσκεται στην μνήμη (**ανοικτά αρχεία, καταχωρητές, program counter κτλ).
- Επομένως οι δυο διεργασίες είναι *σχεδόν* πανομοιότυπες αλλά έχουν i**) διαφορετικό PID** και ii) **διαφορετικό return value από την fork()** (filelocks, alarms, συνέχεια slide 13-18).
- Η εκτέλεση των δυο προγραμμάτων **συνεχίζει από το σημείο στο οποίο έγινε η κλήση της fork()** (δηλαδή το παιδί δεν επαναλαμβάνει όλο το πρόγραμμα αλλά μόνο τις εντολές κάτω από το fork).

Αναπαράσταση στη Μνήμη

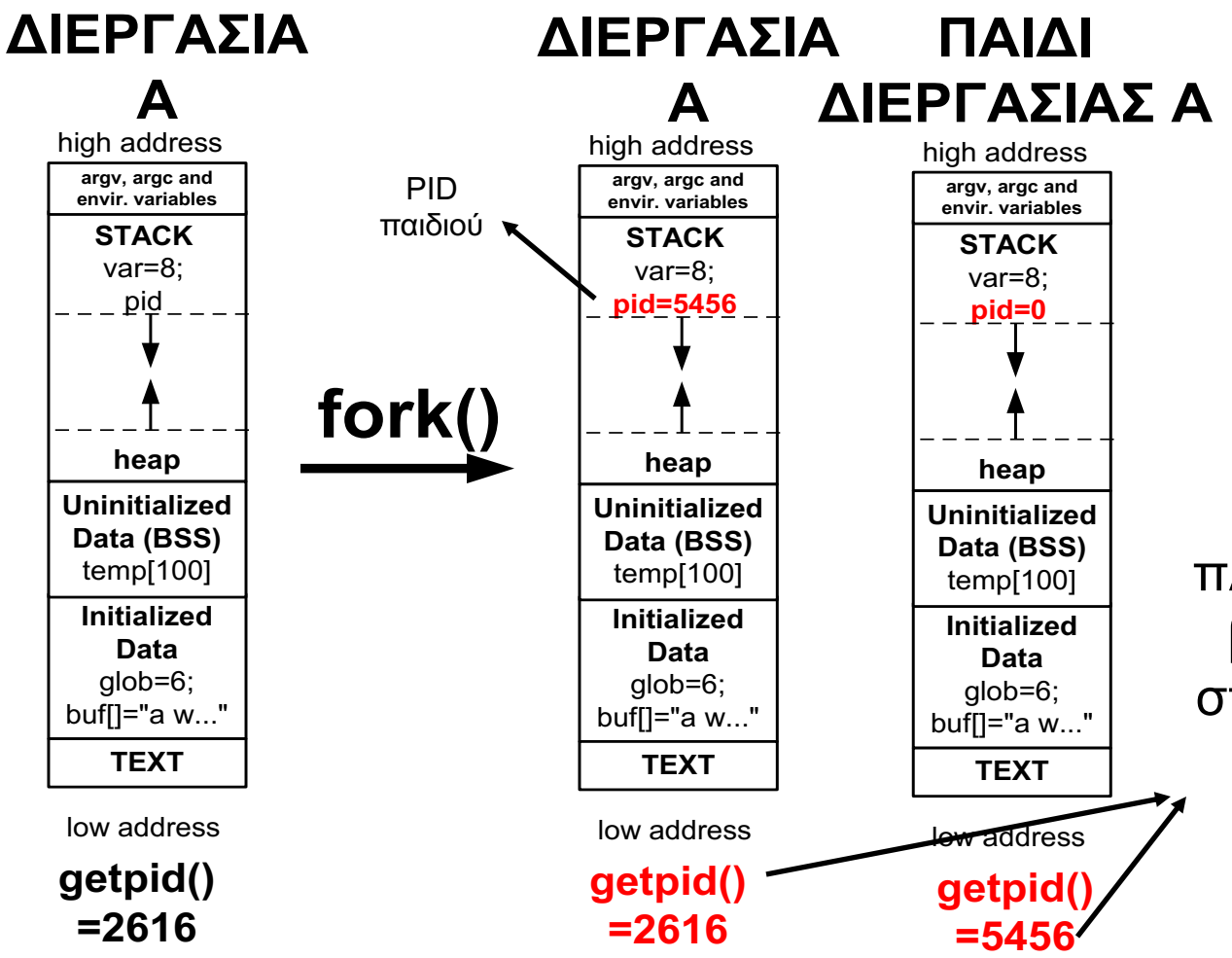


Παράδειγμα 1 - Παρουσίαση fork()

```
#include <unistd.h>  // STDOUT_FILENO
int        glob = 6;                                  /* variable in initialized data */
char buf[] = "a write to stdout\n";                   /* variable in initialized data */
char temp[100];  /* variable in uninitialized data – not used in this program*/

int main(void) {
      int             var = 88;                       /* automatic variable in the stack */
      pid_t           pid;                            /* automatic variable in the stack */
      if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
                      perror("write error");          /* writes to File Descriptor (FD#1) */
      printf("before fork\n");                         /* buffered write to #FD1 */

      if ( (pid = fork()) < 0)  perror("fork error");     /* if fork was unsuccessful */
      else if (pid == 0) { glob++; var++;  }              /* child: modify variables */
      else { sleep(2); }                                  /* parent: sleep for 2 sec. */

      /*  Both Child and Parent execute this, but with different outputs. */
      printf("getpid = %d, glob = %d, var = %d\n", pid, glob, var);
      exit(0);
}
```

Παράδειγμα 1 - fork()

Κλωνοποίηση Διεργασίας με την fork()



Μετά το fork οι δυο διεργασίες είναι πανομοιότυπες, απλά έχουν

i) διαφορετικό PID και **ii) διαφορετικό return value από την fork()**

Παράδειγμα 1 – Εκτέλεση

Πιο κάτω εκτελούμε την fork() και βλέπουμε τα δεδομένα εξόδου.

$ myfork
a write to stdout
before fork
pid=0, **getpid = 3072**, glob = 7, var = 89 **# child's output**
…Καθυστέρηση δυο δευτερολέπτων (από τον πατέρα)
pid=3072, **getpid = 5340**, glob = 6, var = 88 **# parent's output**

ChildID
MyID
MyID
ChildID

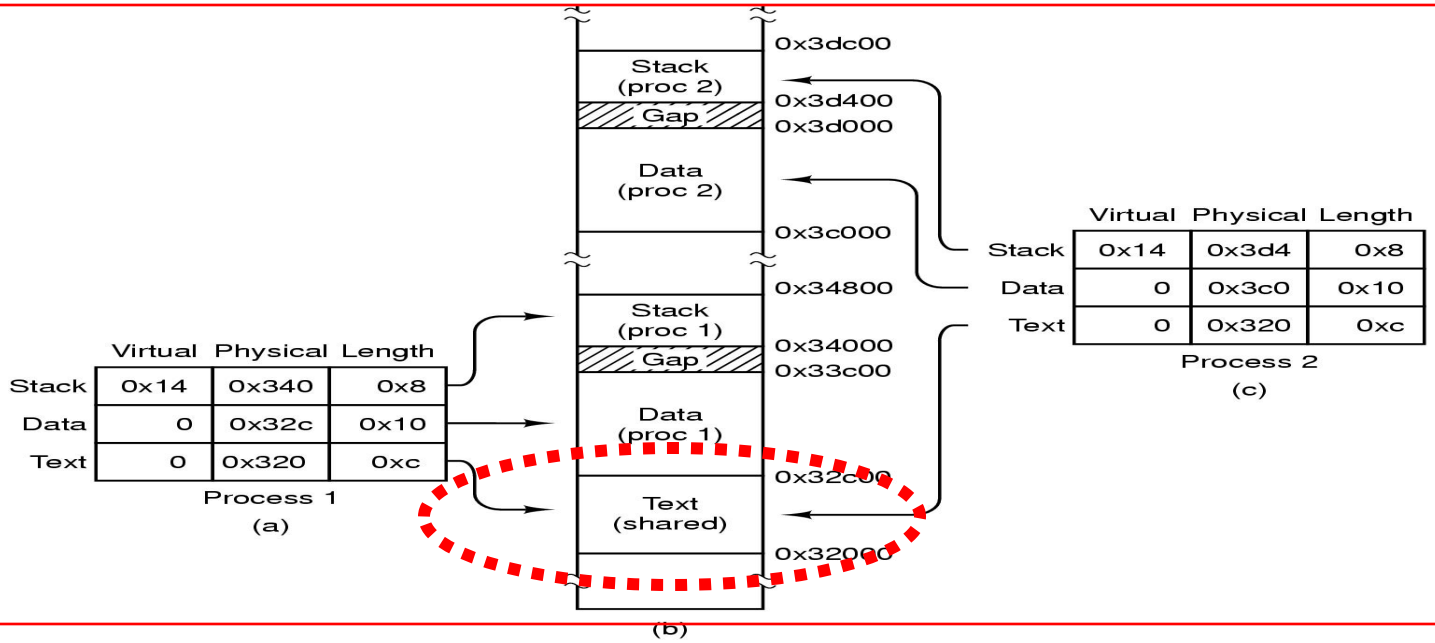
<u>Παρατηρήσεις</u>

- **Δεν μπορούμε να ξέρουμε την σειρά εκτέλεσης (πατέρα-παιδιού),** εάν και σε αυτό το παράδειγμα πάντα προλαβαίνει να ολοκληρώσει το παιδί πρώτο (λόγω της καθυστέρησης 2 δευτερολέπτων στον κώδικα του πατέρα).
- **Για να ορίσουμε την σειρά εκτέλεσης,** μπορούμε να χρησιμοποιήσουμε *i) signals* ή ii) *κάποια μορφή δια-διεργασιακής* επικοινωνίας για να συντονίσουμε την σειρά (θα μελετηθούν στην συνέχεια του μαθήματος).

Δ. Δημιουργία Διεργασιών
Η κλήση συστήματος fork()

Μετά την δημιουργία οι διεργασίες ΔΕΝ ΜΟΙΡΑΖΟΝΤΑΙ πλέον δεδομένα (stack, heap) μεταξύ τους

Για να επικοινωνήσουν οι διεργασίες στην συνέχεια χρειάζονται άλλους μηχανισμούς: IPC (Interprocess Communication), Signals, etc. … θα μελετηθούν στην συνέχεια του μαθήματος.



Η εντολές μηχανής (text) μπορεί ωστόσο να είναι κοινές μεταξύ δυο διεργασιών. i.e., shared object (so) i.e., shared libraries

Δ. Δημιουργία Διεργασιών
Τι δεν κληρονομείται

Τι δεν κληρονομείται στο παιδί με την fork();

- **Η τιμή επιστροφής της fork() (το pid()).**
- **ProcessID,** το παιδί έχει ένα νέο processID. Το **Parent-ProcessID** είναι διαφορετικό.
- **File-Locks,** τα οποία έθεσε ο πατέρας δεν κληρονομούνται (**flock** για πρόγραμμα κελύφους και **fcntl** για την C).
- **Pending Alarms**, αυτά είναι signals τα οποία στέλνονται από τον **πυρήνα** στο **πατέρα περιοδικά** (δεν θα τα λαμβάνει και το παιδί).
  - Αυτά τα alarms θέτονται με την κλήση εντολής alarm() όπως θα δούμε στην επόμενη διάλεξη.
  - Σημειώστε ότι τα περισσότερα από τα πιο πάνω φυλάγονται μέσα στο process table του πυρήνα. Επομένως δεν ενοχλούν κατά την διάρκεια της κλωνοποίησης.

Ε. Μεταβλητές Περιβάλλοντος

Η κλήση βιβλιοθήκης getenv()/putenv()

- Γνωρίζουμε ότι το περιβάλλον κάθε διεργασίας περιέχει διαφορές μεταβλητές (π.χ. PATH, USER, HOME, etc).
- Αυτές οι μεταβλητές περνάνε σε μια διεργασία σαν μέρος του **process image**, κατακρίβειαν ένας πίνακας από pointers σε strings (ένας pointer ανά μεταβλητή)
- Για να έχουμε πρόσβαση σε αυτές τις τιμές χρησιμοποιούμε τις πιο κάτω συναρτήσεις.

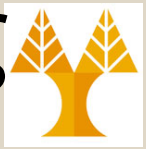
char *getenv(const char *name)

Επιστρέφει: δείκτη στην τιμή της μεταβλητής περιβάλλοντος name ή NULL not found

int putenv(char *str).

Θέτει το ζεύγος «name=value» σαν μεταβλητή περιβάλλοντος. Εάν πετύχει επιστρέφει 0. Η μεταβλητή χάνεται μετά την ολοκλήρωση της διεργασίας.

Ζ. Ορφανές (Orphan) Διεργασίες
Zombie Processes =/= Orphan Processes



Ορφανή Διεργασία

Μια διεργασία Α της οποίας ο γονέας έχει τερματίσει την λειτουργία του.

- Σε αυτή την περίπτωση γονέας της Α γίνεται η διεργασία init (PID#1).
- Ορφανές διεργασίες είναι **φυσιολογική κατάσταση**.
- Κάθε διεργασία χρειάζεται να έχει γονέα για να μπορεί να του παραδώσει το exit code της.

Zombie Διεργασίες

Μια διεργασία Α της οποίας ο **γονέας** δεν έχει αποδεχθεί τον κωδικό εξόδου της Α (πιθανώς επειδή είναι απασχολημένος).

- Για όλο αυτό το διάστημα η διεργασία είναι μια ζωντανή-νεκρή (zombie) διεργασία.
- Οι Zombie διεργασίες είναι **προβληματική κατάσταση** … θα δούμε πως ακριβώς στη συνέχεια.

Ζ. Παράδειγμα 2 - Ορφανή Διεργασία

```
#include <stdio.h> /* For printf*/

int main(void) {
        pid_t           pid;
        printf("before fork\n");

        if ( (pid = fork()) < 0)   perror("fork error");
        else if (pid == 0) {                    /* child */
            sleep(4);           // delay child to create orphan
            printf("child: pid(val)=%d, getpid=%d, getppid=%d \n", pid, getpid(),
        getppid());
        }
        else {
            /* Quit quickly so that the child becomes orphaned. */
            printf("parent: pid(val)=%d, getpid=%d, getppid=%d \n", pid, getpid(),
        getppid());
        }
        exit(0);
}
```

Εδώ θα επιχειρήσουμε να καθυστερήσουμε την εκτέλεση του παιδιού για να ολοκληρώσει ο γονέας και να γίνει ορφανή διεργασία το παιδί.

Σημειώστε ότι μια διεργασία τερματίζει είτε με exit ή return από το main(). Το exit code διοχετεύεται στον πατέρα (ο οποίος το λαμβάνει με την εντολή wait)

7. Παράδειγμα 2 – Εκτέλεση

Πιο κάτω εκτελούμε την myfork2.

ChildID MyID Parent ID

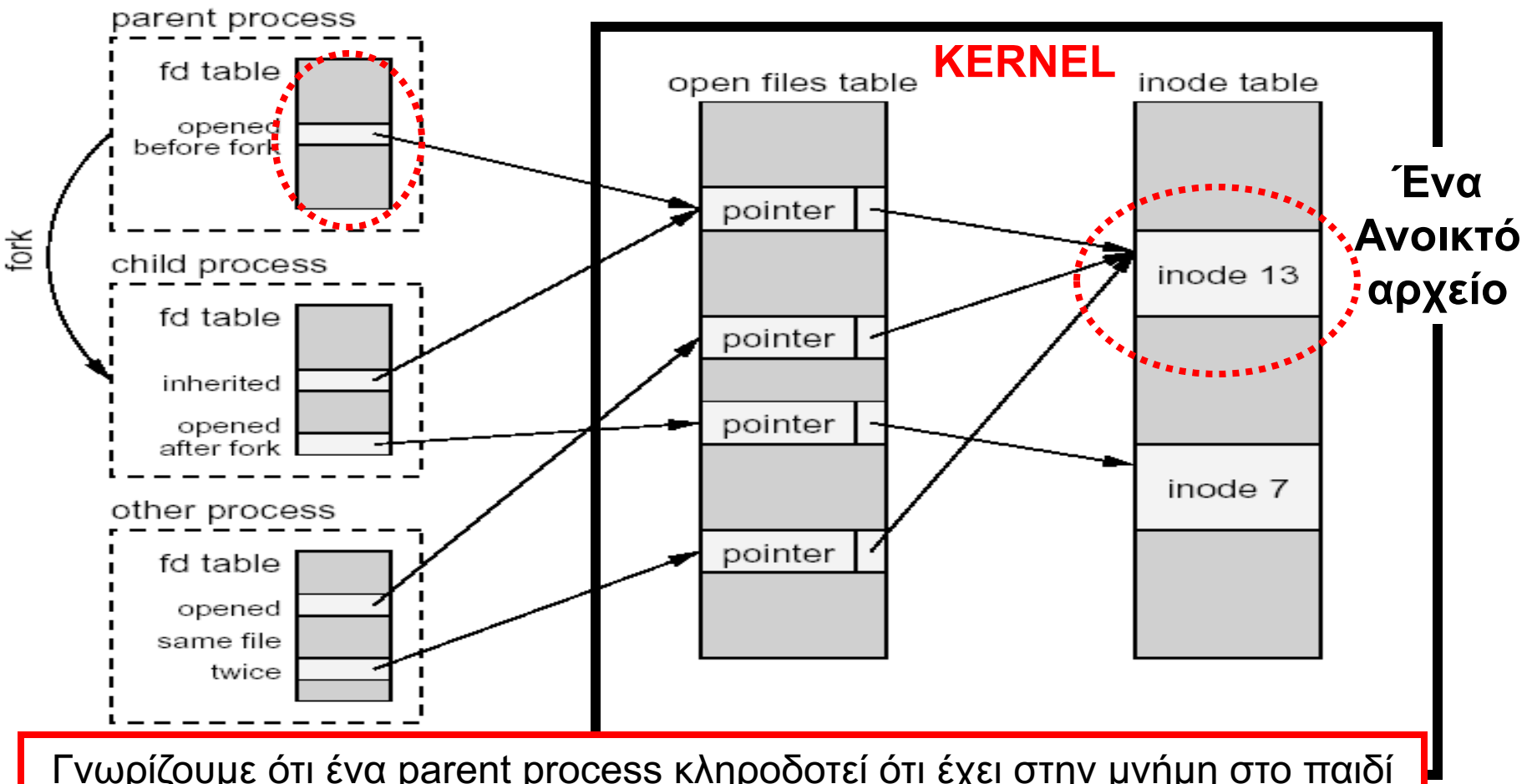
```
$ myfork2
before fork
parent: pid(val)=5224, getpid=4544, getppid=2616
$ # although the command prompt shows up, \
   # after a while we also get
child: pid(val)=0, getpid=5224, getppid=1
```

Πατέρας είναι η διεργασία με PID#1 (η init)

Παρατηρήσεις

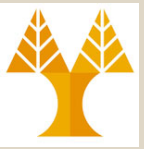
- **Παρατηρούμε ότι το παιδί έχει σαν πατέρα το PID#1,** ουσιαστικά έχει υιοθετηθεί από το init process.
- **Ο Πατέρας/Παιδί έχουν το ίδιο Stdout** (δηλαδή γραφούν και οι δυο στο terminal). Αυτό συμβαίνει παρόλο που τερμάτισε ο πατέρας πριν το παιδί! (Εξήγηση Επόμενη Διαφάνεια).

Η. Διεργασίες και Αρχεία



Γνωρίζουμε ότι ένα parent process κληροδοτεί ότι έχει στην μνήμη στο παιδί του. Μεταξύ άλλων κληροδοτεί και τα ανοικτά αρχεία. Ουσιαστικά γίνεται duplicate το In-Process File Descriptor (FD) Table. Το γεγονός ότι υπάρχει μόνο ένα ανοικτό αρχείο προστατεύει από write-after-write προβλήματα.

Η. Διεργασίες και Αρχεία



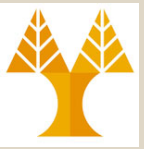
Μέσα στον Πυρήνα

- **Inode Table:** Κάθε αρχείο εμφανίζεται μόνο μια φορά σε αυτό τον πίνακα.
- **Open Files Table:** Μια εγγραφή δημιουργείται όποτε εκτελέσουμε open(). Η εγγραφή περιέχει **i) δείκτη στο inode** table**, ii) offset** μέσα στο αρχείο; **iii) status** (π.χ. read, write, append) με το οποίο ανοίξαμε το αρχείο.

Μέσα στην Διεργασία

- **File Descriptor Table**: Ένα για κάθε διεργασία. Κάθε εγγραφή του, ή οποία αναγνωρίζεται από το fd (0,1,…), περιέχει ένα δείκτη στο open files table. (check /proc/$$/fd/)

Θ. Αναμονή Διεργασιών
Η βιβλιοθήκη συστήματος <sys/wait.h>



- Η συνάρτηση wait() προκαλεί αναμονή μιας διεργασίας μέχρι κάποιο παιδί της να τερματίσει και να επιστρέψει το exit code.

- **#include <sys/wait.h>**
  pid_t wait(int *statloc);
  Επιστρέφει την ταυτότητα του παιδιού που τερμάτισε ή -1 σε περίπτωση λάθους (η διεργασία δεν έχει παιδιά)

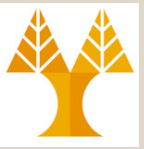
- **Όταν τερματίσει το *statloc περιέχει πληροφορίες για το πώς τερμάτισε (exit code) το παιδί.**
- Σημειώστε ότι το **statloc** είναι **δείκτης σε ακέραιο** και **όχι δομή**. Πρόσβαση στην επιστρεφόμενη τιμή έχουμε μέσω διάφορων macros που είναι ορισμένα στην sys/wait.h.
- Το **statloc>>8** μας δίδει το exit code του παιδιού.
- Επίσης εάν περιμένουμε συγκεκριμένη διεργασία τότε υπάρχει και η εντολή **waitpid(pid_t pid, int *statloc, int options);**
  όπου προσδιορίζουμε το pid της διεργασίας που αναμένουμε

Θ. Αναμονή Διεργασιών

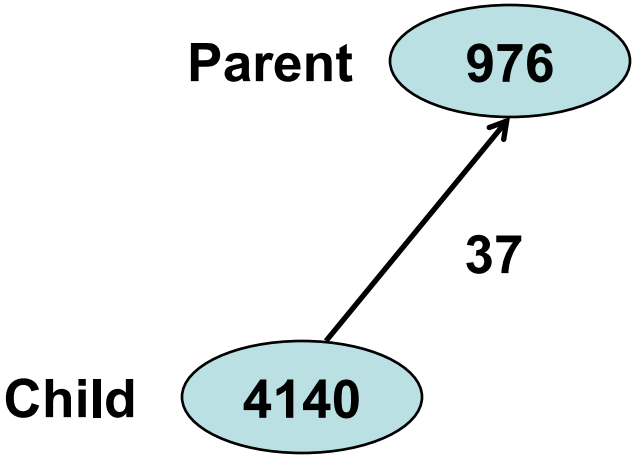
```
int main() {
        int pid;
        int status;
        pid = fork();
        if (pid == -1) {                        /* Check for error */
             perror("fork");  exit(1);
        }
        else if (pid == 0) {    /* The child process */
             printf("Child (%d): Exit \n", getpid());
             exit(37);              /* Exit with a silly number */
        }
        else {
             pid = wait(&status);
             printf("Parent: Child's PID#%d, exit status: %d\n", pid, status>>8);
             printf("Parent (%d): Exit \n", getpid());
        }
}
```

Shifting explanation
Status = 00000000|00000000|00100101|00000000| = 9472
Status>>8 = 00000000|00000000|00000000|00100101| = 37

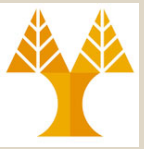
Θ. Αναμονή Διεργασιών



```
$ ./waiting
Child (4140): Exit
Parent: Child's PID#4140, exit status: 37
Parent (976): Exit
```



I. Zombie Διεργασίες

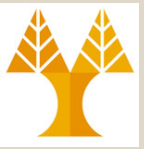


Zombie Διεργασίες

Μια διεργασία Α της οποίας ο γονέας **δεν έχει αποδεχθεί** τον κωδικό εξόδου της Α. Για όλο αυτό το διάστημα η διεργασία Α είναι μια ζωντανή-νεκρή (zombie) διεργασία.

- Η zombie διεργασίες καταναλώνουν άσκοπα μνήμη (είναι εκεί παρόλο που έχουν ολοκληρώσει την λειτουργία τους)

Ι. Παράδειγμα Zombie



```
#include <unistd.h>  // STDOUT_FILENO

int main() {
        int pid;

        pid = fork();
        if (pid == -1) {                    /* Check for error */
             perror("fork");  exit(1);
        }
        else if (pid == 0) {    /* The child process */
             exit(37);                      /* Exit with a silly number */
        }
        else {
             /* The parent process */
             while (1)                      /* Never terminate */
                  sleep(1000);
        }
}
```

Επειδή ο πατέρας δεν τερματίζει ποτέ, το παιδί δεν μπορεί να παραδώσει το exit(37).

Ι. Παράδειγμα Zombie - Εκτέλεση

$./zombie &
[1] 20842

$ ps -a

```
  PID TTY          TIME CMD
 6244 pts/3    00:00:00 su
 6245 pts/3    00:00:00 bash
20842 pts/0    00:00:00 zombie
20843 pts/0    00:00:00 zombie <defunct>
20844 pts/0    00:00:00 ps
```

Η διεργασία παιδί περιμένει τον γονέα να δεχθεί το exit code.

Μέχρι να το δεχθεί είναι zombie process (defunct – νεκρή αλλά καταναλώνει πόρους :-{

$kill -9 **20842** # Κάνωντας kill τον γονέα επιτρέπει στο παιδί να τερματίσει την εκτέλεση του (δίδει το exit code στην init)

$ps –a | grep zombie

Επομένως εδώ δεν επιστρέφεται τίποτα.

Ι. Διόρθωση Zombie

```
#include <unistd.h>  // STDOUT_FILENO

int main() {
        int pid;
        int status;
        pid = fork();
        if (pid == -1) {                        /* Check for error */
             perror("fork");  exit(1);
        }
        else if (pid == 0) {        /* The child process */
             exit(37);              /* Exit with a silly number */
        }
        else {
             pid = wait(&status);
             printf("Exit Code %d %d", pid, status>>8);
              /* The parent process */
             while (1)                          /* Never terminate */
                  sleep(1000);
        }
}
```

Τοποθέτηση wait() πριν τον άπειρο βρόχο στον πατέρα. Αυτό επιτρέπει στο παιδί να παραδώσει το exit() και να μην γίνει zombie.

Αργότερα θα δούμε πως με την χρήση σημάτων SIGCHLD μπορούμε να χειριστούμε αυτή την περίπτωση ασύγχρονα, δηλαδή να προχωρήσει ο γονέας με το while loop

Ι. Διόρθωση Zombie - Εκτέλεση

```
$./notzombie &
[1] 20970
$ Exit Code 20971 37
```

Αυτά εκτυπώνονται από το παιδί, το οποίο έχει τερματίσει.

```
$ ps -a
  PID TTY          TIME CMD
 6244 pts/3    00:00:00 su
 6245 pts/3    00:00:00 bash
20970 pts/0    00:00:00 notzombie
20972 pts/0    00:00:00 ps
```

Τώρα, μόνο ο γονέας συνεχίζει να εκτελείται και δεν έχουμε zombies!