# Διαχείριση διεργασιών

## Linux Process

### Management

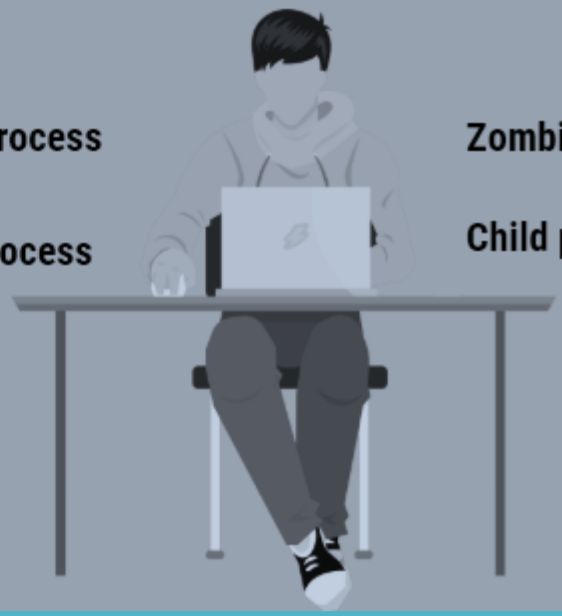
Daemon process

Orphan process

Zombie process

Parent process

Child process



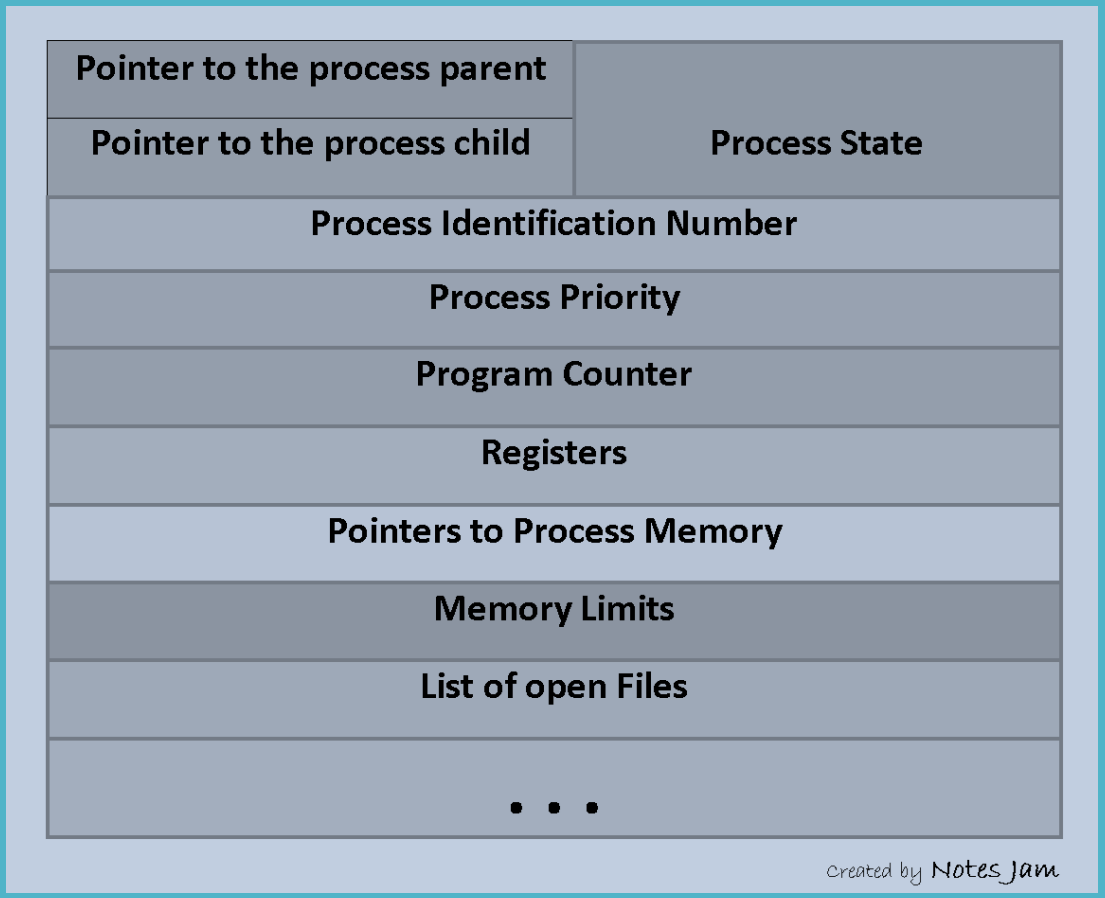
# Διαχείριση διεργασιών

Μιλώντας γενικά, οποιοδήποτε πρόγραμμα εκτελείται σε ένα υπολογιστικό σύστημα, είναι γνωστό ως διεργασία (process). Η κάθε διεργασία εκτελείται ανεξάρτητα από τα άλλα προγράμματα και χρησιμοποιεί τους δικούς της πόρους.

Για κάθε διεργασία που εκτελείται σε ένα υπολογιστικό σύστημα, το λειτουργικό διατηρεί και ενημερώνει συνεχώς μια ειδική δομή δεδομένων που είναι γνωστή ως

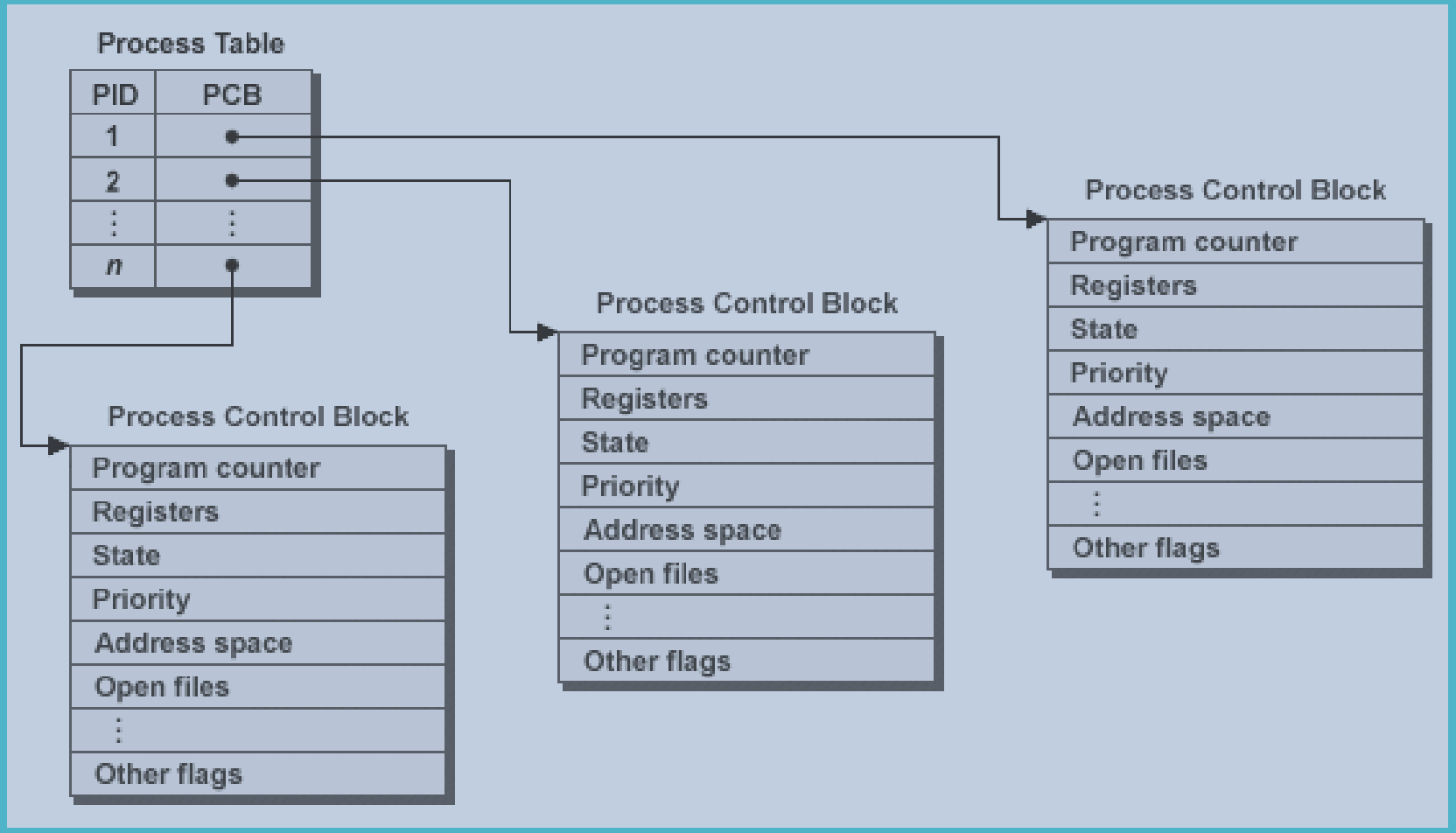
Ομάδα Ελέγχου Διεργασίας

(Process Control Block, PCB)



# Διαχείριση διεργασιών

Το κάθε λειτουργικό σύστημα διατηρεί και ενημερώνει συνεχώς έναν πίνακα διεργασιών (process table) ο οποίος για κάθε διεργασία που εκτελείται στο σύστημα περιέχει το Process Control Block της.



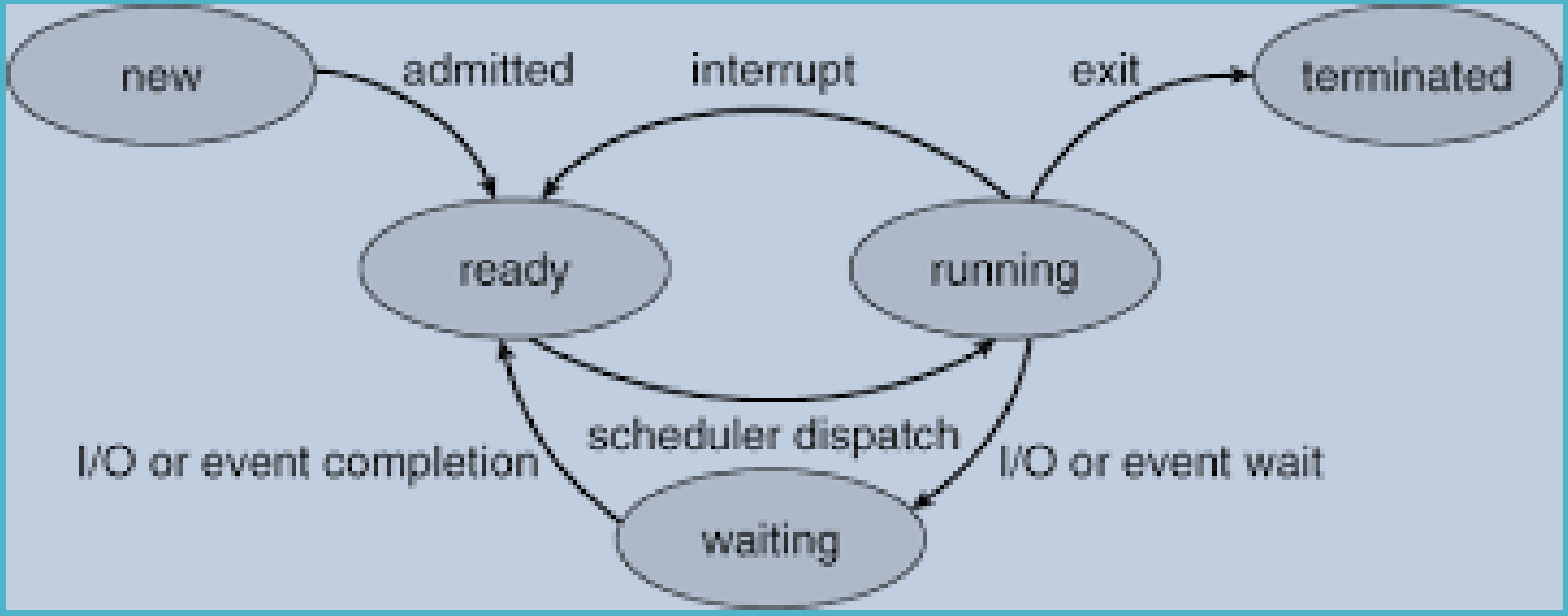
# Διαχείριση διεργασιών

Οι πιο σημαντικές μορφές διαχείρισης διεργασιών από ένα λειτουργικό σύστημα, περιλαμβάνουν:

- Δημιουργία διεργασίας (creation, new process) από το σύστημα ή τους χρήστες.
- Τερματισμός διεργασίας λόγω ολοκλήρωσης (completion) ή κρίσιμου σφάλματος (termination) με ταυτόχρονη αποδέσμευση πόρων.
- Αναστολή διεργασίας (susppend) στα πλαίσια εφαρμογής της εφαρμοζόμενης πολιτικής χρονοπρογραμματισμού.
- Επαναφορά διεργασίας (resume) που έχει ανασταλεί.
- Μεταβολή της τιμής της προτεραιότητας διεργασίας (change process priority).
- Κλείδωμα διεργασίας (block) σε αναμονή κάποιου συμβάντος.
- Ενεργοποίηση κλειδωμένης διεργασίας (wakeup) που την επαναφέρει στην κατάσταση ετοιμότητας.
- Διανομή διεργασίας (dispatch) που τη μεταφέρει από την κατάσταση ετοιμότητας στην κατάσταση εκτέλεσης (δηλαδή της εκχωρεί τον επεξεργαστή για να εκτελεστεί).
- Επικοινωνία με άλλες διεργασίες (inter-process communication).

# Διαχείριση διεργασιών

Κύκλος ζωής διεργασίας σε λειτουργικό σύστημα (process state diagram)




Μεταγωγή περιβάλλοντος (Context Switching)

# Διαχείριση διεργασιών

## Νήματα (Threads)

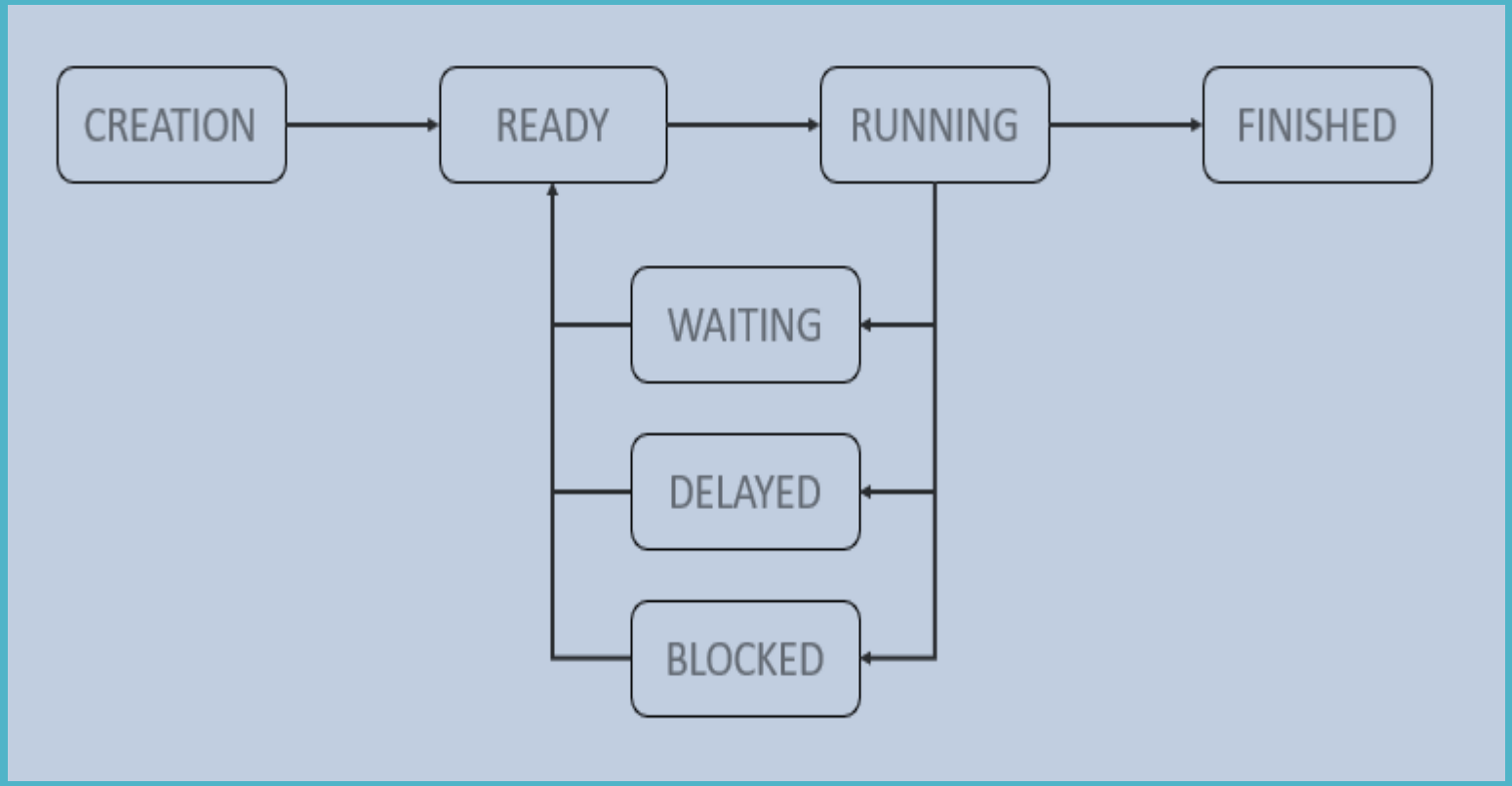
- Τα Threads ξεκινούν από το ίδιο πρόγραμμα, έχουν τα ίδια χαρακτηριστικά που περιγράφουν τις διεργασίες (αρχεία, κάτοχος, ομάδα κατόχου, τρέχων κατάλογος εργασίας, κ.τ.λ.)
- Αποτελούν τη στοιχειώδη μονάδα χρονοπρογραμματισμού στο Linux.
- Σε αυτή την προσέγγιση η διεργασία ορίζεται ως ένα σύνολο από threads που μοιράζονται τους ίδιους πόρους του συστήματος.
- Η μεταγωγή περιβάλλοντος (context switching) είναι γίνεται ανάμεσα σε threads και όχι ανάμεσα σε διεργασίες με αποτέλεσμα να γίνεται πολύ πιο γρήγορα.

Η βιβλιοθήκη pthreads (POSIX threads) <pthread.h>

int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)
void pthread_exit (void *retval)
int pthread_join (pthread_t thread, void **retval)
int pthread_detach (pthread_t thread)
int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)
int pthread_mutex_lock (pthread_mutex_t *mutex)
int pthread_mutex_unlock (pthread_mutex_t *mutex)
int pthread_mutex_destroy (pthread_mutex_t *mutex)

# Διαχείριση διεργασιών

Κύκλος ζωής thread σε λειτουργικό σύστημα (process state diagram)



# Διαχείριση διεργασιών

Στο λειτουργικό σύστημα Linux το Process Control Block υλοποιείται μέσω της δομής task_struct

<linux/sched.h> /home/amarg/focal/kernel/sched/sched.h

**task_struct Struct Reference**

`#include <sched.h>`

Data Fields

| | |
|---|---|
| volatile long | state |
| void * | stack |
| atomic_t | usage |
| unsigned int | flags |
| unsigned int | ptrace |
| int | on_rq |
| int | prio |
| int | static_prio |
| int | normal_prio |
| unsigned int | rt_priority |
| struct sched_class * | sched_class |
| struct sched_entity | se |
| struct sched_rt_entity | rt |
| unsigned char | fpu_counter |
| unsigned int | policy |
| int | nr_cpus_all |
| cpumask_t | cpus_allowe |
| struct list_head | tasks |
| struct mm_struct * | mm |
| struct mm_struct * | active_mm |
| int | exit_state |
| int | exit_code |
| int | exit_signal |
| int | pdeath_signal |
| unsigned int | jobctl |
| unsigned int | personality |
| unsigned | did_exec:1 |
| unsigned | in_execve:1 |
| unsigned | in_iowait:1 |
| unsigned | no_new_privs:1 |
| unsigned | sched_reset_on_fork:1 |
| unsigned | sched_contributes_to_load:1 |
| pid_t | pid |
| pid_t | tgid |
| struct task_struct __rcu * | real_parent |
| struct task_struct __rcu * | parent |
| struct list_head | children |
| struct list_head | sibling |
| struct task_struct * | group_leader |
| struct list_head | ptraced |
| struct list_head | ptrace_entry |
| struct pid_link | pids [PIDTYPE_MAX] |
| struct list_head | thread_group |
| struct completion * | vfork_done |
| int __user * | set_child_tid |
| int __user * | clear_child_tid |
| cputime_t | utime |
| cputime_t | stime |
| cputime_t | utimescaled |
| cputime_t | stimescaled |
| cputime_t | gtime |
| cputime_t | prev_utime |
| cputime_t | prev_stime |
| unsigned long | nvcsw |
| unsigned long | nivcsw |
| struct timespec | start_time |
| struct timespec | real_start_time |
| unsigned long | min_flt |
| unsigned long | maj_flt |
| struct task_cputime | cputime_expires |
| struct list_head | cpu_timers [3] |
| struct cred __rcu * | real_cred |
| struct cred __rcu * | cred |
| char | comm [TASK_COMM_LEN] |
| int | link_count |
| int | total_link_count |
| struct thread_struct | thread |
| struct fs_struct * | fs |
| struct files_struct * | files |
| struct nsproxy * | nsproxy |
| struct signal_struct * | signal |
| struct sighand_struct * | sighand |
| sigset_t | blocked |
| sigset_t | real_blocked |
| sigset_t | saved_sigmask |
| struct sigpending | pending |
| unsigned long | sas_ss_sp |
| size_t | sas_ss_size |
| int(* | notifier )(void *pri |
| void * | notifier_data |
| sigset_t * | notifier_mask |
| struct callback_head * | task_works |
| struct audit_context * | audit_context |
| struct seccomp | seccomp |
| u32 | parent_exec_id |
| u32 | self_exec_id |
| spinlock_t | alloc_lock |
| raw_spinlock_t | pi_lock |
| void * | journal_info |
| struct bio_list * | bio_list |
| struct reclaim_state * | reclaim_state |
| struct backing_dev_info * | backing_dev_info |
| struct io_context * | io_context |
| unsigned long | ptrace_message |
| unsigned long | ptrace_message |
| siginfo_t * | last_siginfo |
| struct task_io_accounting | ioac |
| struct rcu_head | rcu |
| struct pipe_inode_info * | splice_pipe |
| struct page_frag | task_frag |
| int | nr_dirtied |
| int | nr_dirtied_pause |
| unsigned long | dirty_paused_when |
| unsigned long | timer_slack_ns |
| unsigned long | default_timer_slack_ns |

# Διαχείριση διεργασιών

## Η δομή task_struct

```
volatile long state
```

```
#define TASK_RUNNING          0
#define TASK_INTERRUPTIBLE    1
#define TASK_UNINTERRUPTIBLE  2
#define TASK_ZOMBIE           4
#define TASK_STOPPED          8
```

(οι κυριότερες καταστάσεις)

```
int prio
```
dynamic priority considered by the scheduling mechanism

```
int static_prio
```
static priority (assigned during process startup - changed with nice)

```
int normal_prio
```
dynamic priority computed by static priority and scheduling policy

```
unsigned int rt_priority
```
priority for real time processes

```
pid_t pid
```
process ID

```
pid_t tgid
```
thread group ID

```
struct task_struct __rcu * real_parent
```
real parent process

```
struct task_struct __rcu * parent
```
recipient of SIGCHLD

```
struct list_head children
```
list of children processes

```
unsigned int policy
```
scheduling policy

```
struct task_struct *next_task, *prev_task;
struct task_struct *next_run,  *prev_run;
```

```
int exit_state
int exit_code
int exit_signal
```

```
unsigned short uid,euid,suid,fsuid;
unsigned short gid,egid,sgid,fsgid;
```

now in cred struct

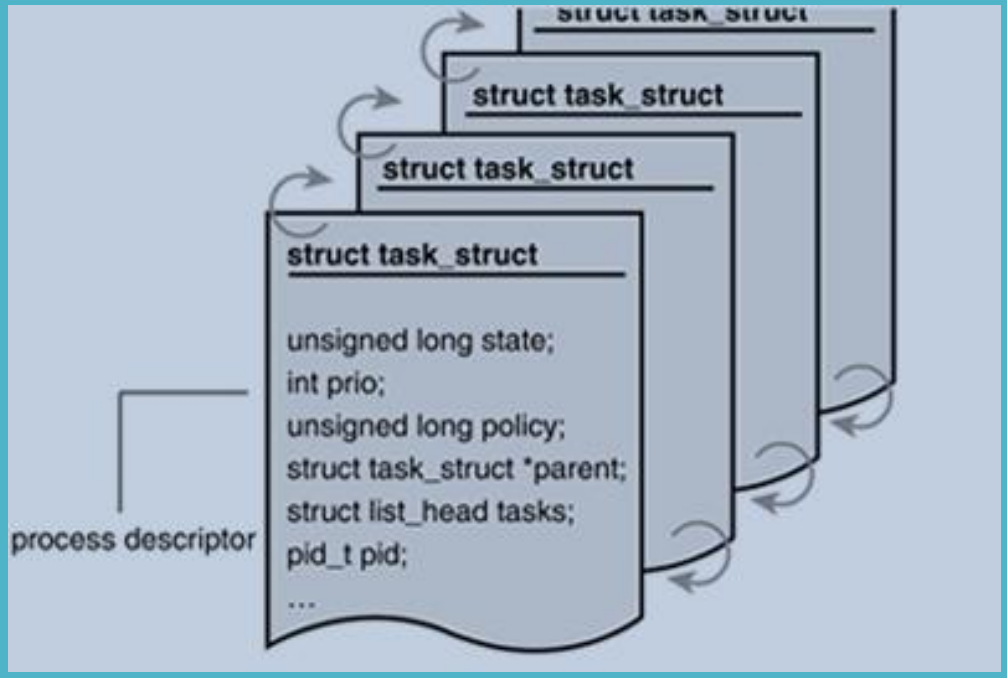
```
struct cred {
        kuid_t          uid;
        kgid_t          gid;
        kuid_t          suid;
        kgid_t          sgid;
        kuid_t          euid;
        kgid_t          egid;
        kuid_t          fsuid;
        kgid_t          fsgid;
```

# Διαχείριση διεργασιών

Στο λειτουργικό σύστημα Linux το Process Control Block υλοποιείται μέσω της δομής task_struct

<linux/sched.h> /home/amarg/focal/kernel/sched/sched.h

(η οποία περιλαμβάνει πάρα πολλά πεδία για να παρουσιαστεί εδώ). Ο πίνακας διεργασιών στο Linux αποτελείται από μία δομή task_struct για κάθε διεργασία που εκτελείται στο σύστημα, με αυτές τις δομές να αποθηκεύονται στους κόμβους μιας κυκλικής διπλής συνδεδεμένης λίστας.



# Διαχείριση διεργασιών

Εκτύπωση πίνακα διεργασιών στο Linux → η εντολή ps

```
amarg@amarg-vbox:~$ ps -elF
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN    RSS PSR STIME TTY          TIME CMD
4 S root           1       0  0  80   0 - 25481 -      11336   0 09:42 ?        00:00:02 /sbin/init splash
1 S root           2       0  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [kthreadd]
1 I root           3       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [rcu_gp]
1 I root           4       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [rcu_par_gp]
1 I root           6       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [kworker/0:0H-kblockd]
1 I root           8       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [mm_percpu_wq]
1 S root           9       2  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [ksoftirqd/0]
1 I root          10       2  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [rcu_sched]
1 S root          11       2  0 -40   - -     0 -          0   0 09:42 ?        00:00:00 [migration/0]
5 S root          12       2  0   9   - -     0 -          0   0 09:42 ?        00:00:00 [idle_inject/0]
1 S root          14       2  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [cpuhp/0]
5 S root          15       2  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [kdevtmpfs]
1 I root          16       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [netns]
1 S root          17       2  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [rcu_tasks_kthre]
1 S root          18       2  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [kauditd]
1 S root          19       2  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [khungtaskd]
1 S root          20       2  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [oom_reaper]
1 I root          21       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [writeback]
1 S root          22       2  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [kcompactd0]
1 S root          23       2  0  85   5 -     0 -          0   0 09:42 ?        00:00:00 [ksmd]
1 S root          24       2  0  99  19 -     0 -          0   0 09:42 ?        00:00:00 [khugepaged]
1 I root          70       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [kintegrityd]
1 I root          71       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [kblockd]
1 I root          72       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [blkcg_punt_bio]
1 I root          73       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [tpm_dev_wq]
1 I root          74       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [ata_sff]
1 I root          75       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [md]
1 I root          76       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [edac-poller]
1 I root          77       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [devfreq_wq]
1 S root          78       2  0 -40   - -     0 -          0   0 09:42 ?        00:00:00 [watchdogd]
1 S root          81       2  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [kswapd0]
1 S root          82       2  0  80   0 -     0 -          0   0 09:42 ?        00:00:00 [ecryptfs-kthrea]
1 I root          84       2  0  60 -20 -     0 -          0   0 09:42 ?        00:00:00 [kthrotld]
```

# Διαχείριση διεργασιών

## Χαρακτηριστικά διεργασιών (pid & ppid)

Process Id (pid) → Θετικός ακέραιος που προσδιορίζει την κάθε διεργασία με μοναδικό τρόπο μέσα στο σύστημα (τύπου pid_t → int).

Parent Process Id (ppid) → Θετικός ακέραιος που προσδιορίζει την γονική της κάθε διεργασίας με μοναδικό τρόπο μέσα στο σύστημα (τύπου pid_t →int).

Ο τύπος pid_t είναι int που ορίζεται στο αρχείο types.h ως **typedef int pid_t;**

Για την ανάκτηση του pid και του ppid για την κάθε διεργασία χρησιμοποιούνται οι συναρτήσεις

```
pid_t getpid ();
pid_t getppid ();
```

που είναι δηλωμένες στο αρχείο επικεφαλίδας unistd.h.

# Διαχείριση διεργασιών

## Χαρακτηριστικά διεργασιών (uid & gid)

User Id (uid) (θετικός ακέραιος) → Ο κωδικός του χρήστη που δημιουργεί τη διεργασία. Εάν είναι uid = 0 τότε ο εν λόγω χρήστης είναι ο διαχειριστής του συστήματος (root) ο οποίος δεν υπόκειται σε έλεγχο ασφάλειας από τον πυρήνα του λειτουργικού. Οι κωδικοί των χρηστών είναι αποθηκευμένοι στο αρχείο συστήματος /etc/passwd.

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
```

Group Id (gid) (θετικός ακέραιος) → Ο κωδικός της πρωτεύουας ομάδας του παραπάνω χρήστη. Οι κωδικοί των ομάδων χρηστών είναι αποθηκευμένοι στο αρχείο /etc/groups.

```
root:x:0:
daemon:x:1:
bin:x:2:
sys:x:3:
adm:x:4:syslog,amarg
tty:x:5:syslog
disk:x:6:
lp:x:7:
mail:x:8:
news:x:9:
uucp:x:10:
man:x:12:
proxy:x:13:
kmem:x:15:
dialout:x:20:
fax:x:21:
voice:x:22:
cdrom:x:24:amarg
floppy:x:25:
tape:x:26:
sudo:x:27:amarg
audio:x:29:pulse
dip:x:30:amarg
www-data:x:33:
```

# Διαχείριση διεργασιών

## Χαρακτηριστικά διεργασιών (uid & gid)

Η χρήση ενός και μοναδικού gid για τις διεργασίες με το ίδιο uid δημιουργούσε προβλήματα, εάν ο ίδιος χρήστης έπρεπε να προσπελάσει αρχεία με διαφορετικό gid.

Για το λόγο αυτό σε μεταγενέστερες υλοποιήσεις μία διεργασία μπορούσε να συσχετιστεί το πολύ με NGROUP διαφορετικά gid όπου η σταθερά NGROUP = 32 δηλώνεται στο <param.h>.

Ορισμός συνόλου gids για την κάθε διεργασία

int setgroups (size_t num, const gid_t * groupList)

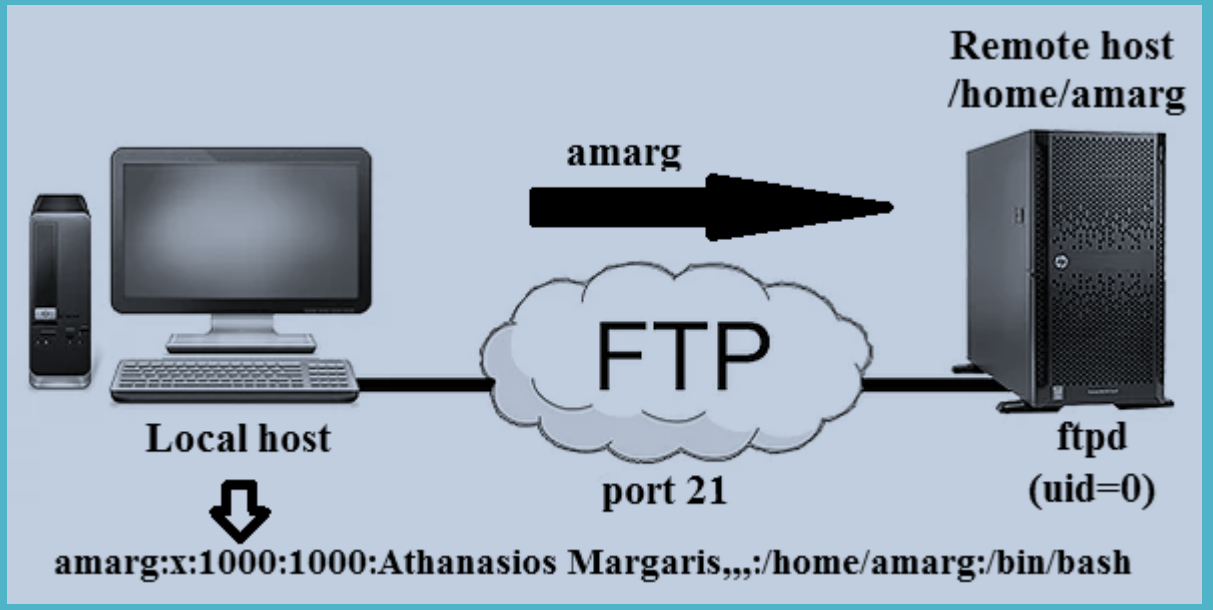
Ανάκτηση συνόλου gids για την κάθε διεργασία

int getgroups (size_t num, gid_t * groupList)

# Διαχείριση διεργασιών

## Αλλαγή του uid και οι εφαρμογές setuid

Σε ορισμένες περιπτώσεις η επιτυχής εκτέλεση μιας διαδικασίας απαιτεί τη χρήση της τιμής uid ενός διαφορετικού χρήστη – παράδειγμα η εφαρμογή ftpd.




Το uid του ftpd τίθεται στην τιμή 1000 (συνήθως ξεκινά νέο αντίγραφο της διεργασίας)
Ωστόσο εάν απαιτηθεί ξανά να είναι uid = 0 αυτό είναι αδύνατο αφού προφανώς μία διεργασία δεν μπορεί να χορηγήσει στον εαυτό της δικαιώματα διαχειριστή.

# Διαχείριση διεργασιών

## Αλλαγή του uid και οι εφαρμογές setuid

ΛΥΣΗ → Χρησιμοποιούνται τρία διαφορετικά uid !

- Real uid (ruid) → πραγματικό uid
- Saved uid (suid) → αποθηκευμένο uid
- Effective uid (euid) → ενεργό uid

Για την περίπτωση του ftpd
Αρχικά ruid = suid = euid = 0 (root access)
Όταν συνδεθεί ο χρήστης με uid = ID τότε
euid = ID ενώ ruid = suid = 0
Όταν απαιτείται root access γίνεται euid = 0
και μετά την ολοκλήρωση ξανά euid = ID

```
cred {
kuid_t          uid;
kgid_t          gid;
kuid_t          suid;
kgid_t          sgid;
kuid_t          euid;
kgid_t          egid;
kuid_t          fsuid;
kgid_t          fsgid;
```

Στη γενική περίπτωση κατά το login του χρήστη θέτουμε ruid = suid = euid = ID < /etc/passwd
Όταν απαιτείται αλλαγή του uid τα ruid και euid καθορίζουν εάν αυτό που ζητά ο χρήστης είναι επιτρεπτό και εάν ναι, θέτουμε euid = ruid ή euid = suid. Εάν είναι euid = 0 (root access) το euid μπορεί να λάβει οποιαδήποτε τιμή.

# Διαχείριση διεργασιών

## Αλλαγή του uid και οι εφαρμογές setuid

Τα στοιχεία των χρηστών είναι αποθηκευμένα στο αρχείο συστήματος /etc/passwd

-rw-r--r-- 1 root root 2797 Σεπ 19 20:17 /etc/passwd

Παρατηρούμε πως ο κάτοχος του αρχείου είναι ο διαχειριστής (root) ο οποίος έχει δικαίωμα ανάγνωσης και εγγραφής (r w -) ενώ οι άλλοι χρήστες έχουν δικαίωμα μόνο ανάγνωσης (r - -).

Ωστόσο, ο κάθε χρήστης χρησιμοποιώντας την εντολή passwd μπορεί να αλλαξει τον κωδικό πρόσβασης στο σύστημα, με το νέο κωδικό να αποθηκεύεται στο αρχείο /etc/passwd αντικαθιστώντας τον παλαιό!

Πως είναι δυνατή η τροποποίηση του περιεχομένου ενός αρχείου, στο οποίο ο χρήστης δεν έχει δικαίωμα εγγραφής?

# Διαχείριση διεργασιών

## Αλλαγή του uid και οι εφαρμογές setuid

Η απάντηση βρίσκεται στη μάσκα δικαιωμάτων της εφαρμογής /usr/bin/passwd

```
amarg@amarg-vbox:~$ which passwd
/usr/bin/passwd
amarg@amarg-vbox:~$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 68208 Μαϊ  28 09:37 /usr/bin/passwd
```

Το bit s (setuid bit) επιτρέπει την εκτέλεση του προγράμματος passwd από οποιονδήποτε χρήστη με τα δικαιώματα που έχει ο κάτοχος του προγράμματος. Αυτός ο κάτοχος είναι ο root με δικαιώματα διαχειριστή και για το λόγο αυτό είναι δυνατή η τροποποίηση του αρχείου /etc/passwd.

Βέβαια θα πρέπει ταυτόχρονα το ruid να ταυτίζεται με το uid της γραμμής του /etc/passwd που πρόκειται να μεταβληθεί, αλλιώς ο κάθε χρήστης θα μπορούσε να αλλάξει το password οποιουδήποτε άλλου χρήστη !!

Οι εφαρμογές αυτού του τύπου που όταν εκτελούνται έχουν το ίδιο euid (ή egid) με τον κάτοχο του προγράμματος ανεξάρτητα από το ποιος χρήστης τις εκτελεί λέγονται εφαρμογές τύπου setuid ή setgid.

# Διαχείριση διεργασιών

## Αλλαγή του uid και οι εφαρμογές setuid

Εάν ID ειναι το uid του χρήστη που χρησιμοποιεί μία εφαρμογή setuid αυτό αποθηκεύεται στο euid η παλαιά τιμή του οποίου απποθηκεύεται στο suid έτσι ώστε να μπορεί να ανακτηθεί αργότερα.

Άρα για την εκτέλεση αυτών των προγραμμάτων γίνονται οι εκχωρήσεις τιμών

suid = euid και euid = ID

ενώ στο τέλος το euid ανακτά την αρχική του τιμή ως euid = suid. Επίσης μπορούμε να θέσουμε ruid = ID

Οι σχετικές κλήσεις συστήματος

| | |
|---|---|
| setuid | set real user ID |
| getuid | get real user ID |
| geteuid | get effective user ID |
| getegid | get effective group ID |
| setreuid | set real and effective user IDs |
| setregid | set real and effective group IDs |
| setfsuid | set user identity used for file system checks |
| setfsgid | set group identity used for file system checks |
| setresuid | set real, effective and saved user or group ID |
| getresuid | get real, effective and saved user or group ID |

```
int   setuid   (uid_t);
int   setgid   (gid_t);
int   seteuid  (uid_t);
int   setegid  (gid_t);
int   setreuid (uid_t, uid_t);
int   setregid (gid_t, gid_t);
uid_t getuid   (void);
gid_t getgid   (void);
uid_t geteuid  (void);
gid_t getegid  (void);
```

# Διαχείριση διεργασιών

## setuid, setgid & sticky bits

setuid bit → εάν αυτό το bit είναι on η εφαρμογή δεν εκτελείται με τα δικαιώματα του χρήστη που την εκτελεί, αλλά με τα δικαιώματα του κατόχου της. Δεν επηρεάζει τα δικαιώματα των καταλόγων και απαιτεί την ενεργοποίηση του bit x στην τριάδα bits του κατόχου.

setgid bit → εάν αυτό το bit είναι on η εφαρμογή δεν εκτελείται με τα δικαιώματα της ομάδας του χρήστη που την εκτελεί, αλλά με τα δικαιώματα της ομάδας του κατόχου της. Εάν χρησιμοποιηθεί με καταλόγους, όλα τα νέα αρχεία που δημιουργούνται μέσα σε αυτούς λαμβάνουν τα δικαιώματα του γονικού καταλόγου. Απαιτεί την ενεργοποίηση του bit x στην τριάδα bits της ομάδας του κατόχου.

sticky bit → επιδρά μόνο σε καταλόγους και εάν ενεργοποιηθεί, τότε όλα τα αρχεία που υπάρχουν μέσα σε αυτούς μπορούν να τροποποιηθούν μόνο από τους κατόχους τους (π.χ. ο κατάλογος /tmp).

# Διαχείριση διεργασιών

## setuid, setgid & sticky bits


Με ποιο τρόπο γίνεται η ενεργοποίηση των τριών ειδικών bits

Ενεργοποίηση setuid bit → 4
Ενεργοποίηση setguid bit → 2
Ενεργοποίηση sticky bit → 1

Αυτός ο αριθμός γράφεται πριν από την οκταδική μάσκα δικαιωμάτων της chmod η οποία πλέον έχει τέσσερα ψηφία

# Διαχείριση διεργασιών

## setuid, setgid & sticky bits

Το γράμμα που εκτυπώνεται (s ή t) είναι κεφαλαίο ή μικρό ανάλογα με το εάν το bit εκτέλεσης x έχει ενεργοποιηθεί ή όχι, αντίστοιχα

### Παραδείγματα ορισμού setuid bit

chmod **4**544 file1 ( r – x r - - r - - ) → – r – s r - - r - - 1 amarg amarg 0 Σεπ 24 23:02 file1
chmod **4**644 file1 ( r w – r - - r - - ) → - r w S r - - r - - 1 amarg amarg 0 Σεπ 24 23:02 file1

### Παραδείγματα ορισμού setgid bit

chmod **2**656 file1 ( r w – r – x r w - ) → - r w – r – s r w - 1 amarg amarg 0 Σεπ 24 23:02 file1
chmod **2**665 file1 ( r w – r w – r - - ) → - r w – r w S r - x 1 amarg amarg 0 Σεπ 24 23:02 file1

### Παραδείγματα ορισμού sticky bit

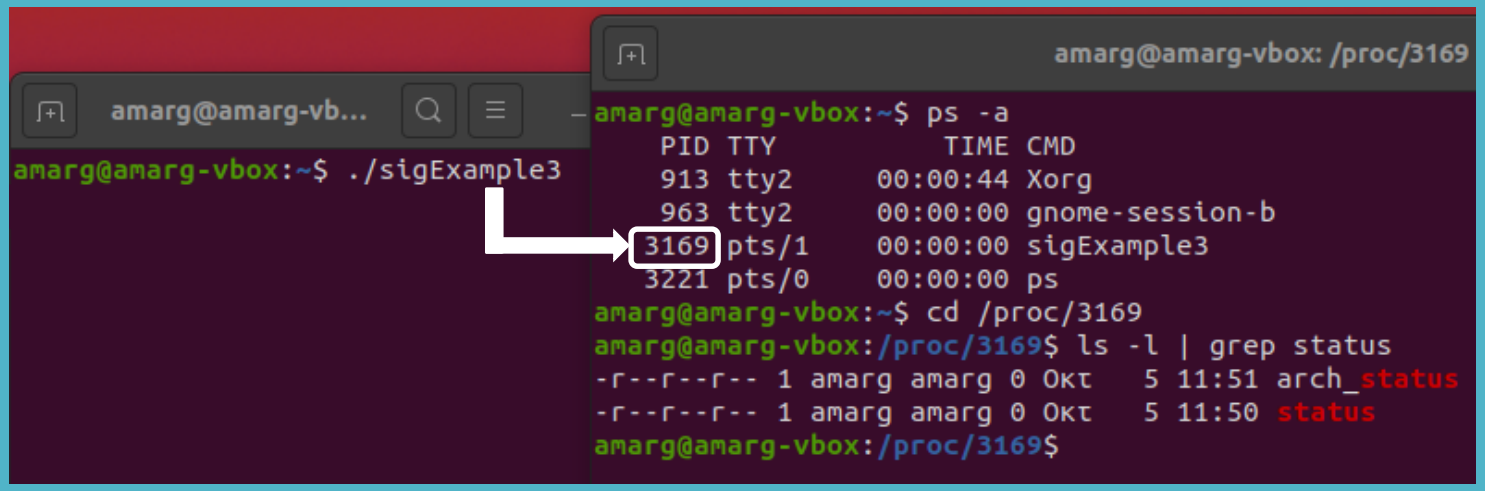
chmod **1**755 file1 ( r w x r – x r – x ) → - r w x r – x r - t 1 amarg amarg 0 Σεπ 24 23:02 file1
chmod **1**756 file1 ( r w x r – x r w - ) → -r w x r – x r w T 1 amarg amarg 0 Σεπ 24 23:02 file1

**Απενεργοποίηση των setuid, setgid και sticky bits** chmod **0**544 file1 **ή πιο απλά** chmod 544 file1

# Διαχείριση διεργασιών

Ανάκτηση των πληροφοριών που σχετίζονται με την κατάσταση της διεργασίας

Για κάθε διεργασία με pid xyzt, το αρχείο /proc/xyzt/status περιέχει πληροφορίες για την κατάσταση εκτέλεσης της διεργασίας.




Αυτό το αρχείο περιέχει όλες τις πληροφορίες που εκτυπώνονται από την εντολή ps (στην πραγματικότητα η εντολή ps ανατρέχει στο proc file system για να ανακτήσει τις πληροφορίες που εκτυπώνει) αλλά και άλλες πληροφορίες που δεν εκτυπώνει η ps

# Διαχείριση διεργασιών

Παράδειγμα περιεχομένων status file

```
Name:   rsyslogd
Umask:  0022
State:  S (sleeping)
Tgid:   373
Ngid:   0
Pid:    373
PPid:   1
TracerPid:      0
Uid:    104     104     104     104
Gid:    110     110     110     110
FDSize:         64
Groups:         4 5 110
NStgid:         373
NSpid:  373
NSpgid:         373
NSsid:  373
VmPeak:           224332 kB
VmSize:           224332 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:      4792 kB
VmRSS:      4792 kB
RssAnon:            1040 kB
RssFile:            3752 kB
RssShmem:              0 kB
VmData:            18304 kB
VmStk:       132 kB
VmExe:       468 kB
VmLib:      3568 kB
VmPTE:        80 kB
VmSwap:                0 kB
HugetlbPages:          0 kB
CoreDumping:    0
THP_enabled:    1
Threads:        4
```

```
SigQ:   0/7730
SigPnd:         0000000000000000
ShdPnd:         0000000000000000
SigBlk:         0000000000000000
SigIgn:         0000000001001206
SigCgt:         0000000180314001
CapInh:         0000000000000000
CapPrm:         0000000000000000
CapEff:         0000000000000000
CapBnd:         0000003fffffffff
CapAmb:         0000000000000000
NoNewPrivs:     0
Seccomp:        0
Speculation_Store_Bypass:       vulnerable
Cpus_allowed:   1
Cpus_allowed_list:      0
Mems_allowed:
00000000,00000000,00000000,00000000,00000000
Mems_allowed_list:      0
voluntary_ctxt_switches:        51
nonvoluntary_ctxt_switches:     13
```

# Διαχείριση διεργασιών

Οι πληροφορίες που αποθηκεύονται στο status file

| Field | Content |
|---|---|
| Name | filename of the executable |
| Umask | file mode creation mask |
| State | state (R is running, S is sleeping, D is sleeping in an uninterruptible wait, Z is zombie, T is traced or stopped) |
| Tgid | thread group ID |
| Ngid | NUMA group ID (0 if none) |
| Pid | process id |
| PPid | process id of the parent process |
| TracerPid | PID of process tracing this process (0 if not) |
| Uid | Real, effective, saved set, and file system UIDs |
| Gid | Real, effective, saved set, and file system GIDs |
| FDSize | number of file descriptor slots currently allocated |
| Groups | supplementary group list |
| NStgid | descendant namespace thread group ID hierarchy |
| NSpid | descendant namespace process ID hierarchy |
| NSpgid | descendant namespace process group ID hierarchy |
| NSsid | descendant namespace session ID hierarchy |
| VmPeak | peak virtual memory size |
| VmSize | total program size |
| VmLck | locked memory size |
| VmPin | pinned memory size |
| VmHWM | peak resident set size (“high water mark”) |

# Διαχείριση διεργασιών

Οι πληροφορίες που αποθηκεύονται στο status file

| | |
|---|---|
| VmRSS | size of memory portions. It contains the three following parts (VmRSS = RssAnon + RssFile + RssShmem) |
| RssAnon | size of resident anonymous memory |
| RssFile | size of resident file mappings |
| RssShmem | size of resident shmem memory (includes SysV shm, mapping of tmpfs and shared anonymous mappings) |
| VmData | size of private data segments |
| VmStk | size of stack segments |
| VmExe | size of text segment |
| VmLib | size of shared library code |
| VmPTE | size of page table entries |
| VmSwap | amount of swap used by anonymous private data (shmem swap usage is not included) |
| HugetlbPages | size of hugetlb memory portions |
| CoreDumping | process's memory is currently being dumped (killing the process may lead to a corrupted core) |
| THP_enabled | process is allowed to use THP (returns 0 when PR_SET_THP_DISABLE is set on the process |
| Threads | number of threads |
| SigQ | number of signals queued/max. number for queue |
| SigPnd | bitmap of pending signals for the thread ← |
| ShdPnd | bitmap of shared pending signals for the process ← |
| SigBlk | bitmap of blocked signals ← |
| SigIgn | bitmap of ignored signals ← |
| SigCgt | bitmap of caught signals ← |
| CapInh | bitmap of inheritable capabilities |

# Διαχείριση διεργασιών

Οι πληροφορίες που αποθηκεύονται στο status file

| | |
|---|---|
| CapPrm | bitmap of permitted capabilities |
| CapEff | bitmap of effective capabilities |
| CapBnd | bitmap of capabilities bounding set |
| CapAmb | bitmap of ambient capabilities |
| NoNewPrivs | no_new_privs, like prctl(PR_GET_NO_NEW_PRIV, …) |
| Seccomp | seccomp mode, like prctl(PR_GET_SECCOMP, …) |
| Speculation_Store_Bypass | speculative store bypass mitigation status |
| Cpus_allowed | mask of CPUs on which this process may run |
| Cpus_allowed_list | Same as previous, but in “list format” |
| Mems_allowed | mask of memory nodes allowed to this process |
| Mems_allowed_list | Same as previous, but in “list format” |
| voluntary_ctxt_switches | number of voluntary context switches |
| nonvoluntary_ctxt_switches | number of non voluntary context switches |

# Διαχείριση διεργασιών

## Δημιουργία διεργασιών – η εντολή fork ()

Η δημιουργία διεργασιών στο λειτουργικό σύστημα Linux πραγματοποιείται με την εντολή

pid_t fork (void);

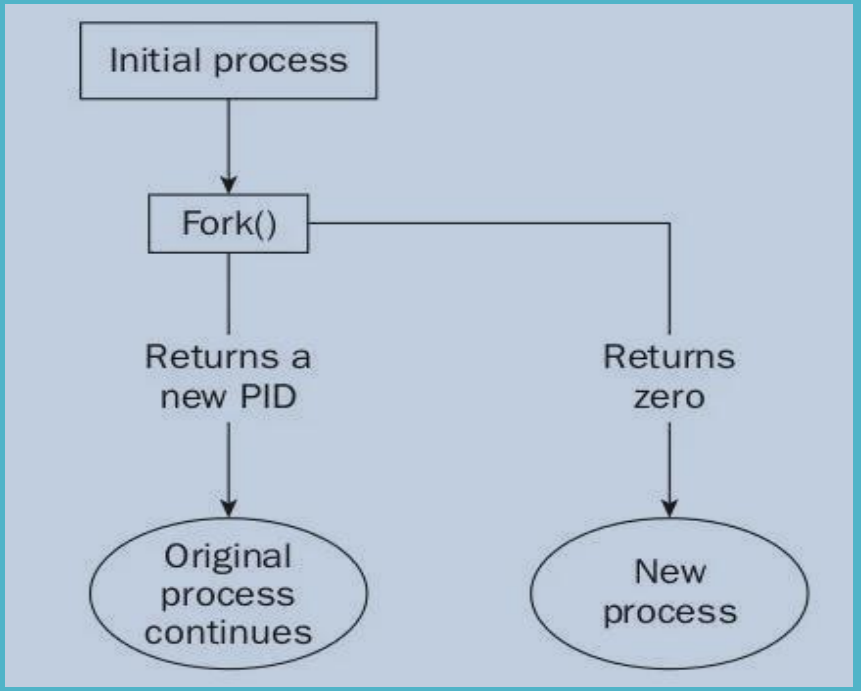
Η νέα διεργασία που δημιουργείται με τον τρόπο αυτό αποτελεί θυγατρική διεργασία της διεργασίας που κάλεσε τη fork και ως εκ τούτου οι δύο διεργασίες σχετίζονται με σχέση γονέα – παιδιού.

Η fork επιστρέφει την τιμή της τόσο στη γονική όσο και στη θυγατρική διεργασία !!

- Η τιμή που επιστρέφεται στη γονική διεργασία είναι το pid της θυγατρικής διεργασίας.
- Η τιμή που επιστρέφεται στη θυγατρική διεργασία είναι η τιμή 0.

Με τον τρόπο αυτό είναι δυνατή η διάκριση ανάμεσα στις δύο διεργασίες.

Μία επιστρεφόμενη τιμή ίση με -1 υποδηλώνει την εμφάνιση σφάλματος δημιουργίας της διεργασίας.



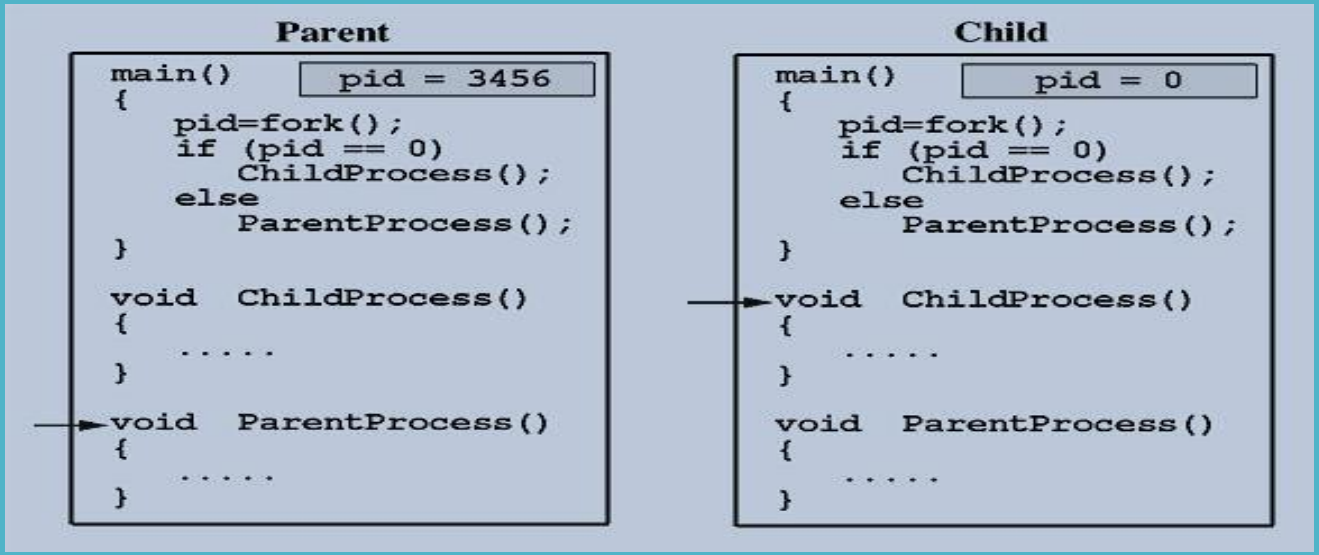
# Διαχείριση διεργασιών

## Δημιουργία διεργασιών – η εντολή fork ()

Η γονική και η θυγατρική διεργασία έχουν διαφορετικό χώρο διευθύνσεων μνήμης αλλά με το ίδιο περιεχόμενο αφού εκτελείται ο ίδιος κώδικας.

Μετά την κλήση της fork () οι δύο διεργασίες αρχίζουν να εκτελούνται και υφίστανται ταυτόχρονα.

Οι μεταβλητές που έχουν αρχικοποιηθεί πριν τη κλήση της fork () έχουν τις ίδιες τιμές στις δύο διεργασίες ενώ από εκεί και πέρα οι τροποποιήσεις που πραγματοποιούνται στη μεταβλητή της μίας διεργασίας δεν επηρεάζουν την αντίστοιχη μεταβλητή της άλλης διεργασίας.



# Διαχείριση διεργασιών

## Τερματισμός διεργασιών – η εντολή exit ()

Μία διεργασία μπορεί να τερματιστεί με δύο διαφορετικούς τρόπους και πιο συγκεκριμένα:

- Φυσιολογικά, όταν ολοκληρωθεί ή καλώντας την εντολή exit ή την εντολή _exit ()
- Απροσδόκητα, όταν δεχθεί σήμα τερματισμού (kill) ή όταν καταρρεύσει το σύστημα.

Υπάρχουν δύο συναρτήσεις για κανονικό τερματισμό, η **void _exit (int exitCode)** η οποία είναι κληση συστήματος και η **void exit (int exitCode)** η οποία είναι συνάρτηση της C. Στη δεύτερη περίπτωση, η συνάρτηση exit καλεί τις συναρτήσεις που έχουν οριστεί ως

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

και ύστερα καλεί τη συνάρτηση _exit(). Ο απροσδόκητος τερματισμός μιας διεργασίας γίνεται από την

**int kill (pid_t pid, int sig)**

όπου pid ο κωδικός της διεργασίας προς τερματισμό και sig κατάλληλο σήμα τερματισμού. Εάν είναι pid > 0 το σήμα στέλνεται στη διεργασία με αυτόν τον κωδικό, ενώ εάν είναι pid = -1 το σήμα στέλνεται σε όλες τις διεργασίες εκτός από την init κατά το shutdown του συστήματος.

Το λειτουργικό διαθέτει την εντολή **kill pid** όπου pid ο κωδικός της διεργασίας προς τερματισμό

# Διαχείριση διεργασιών

## Αναστολή διεργασιών – η εντολή wait4

Στενά συνδεδεμένη με τις συναρτήσεις fork και exit είναι η συνάρτηση

pid_t wait4 (pid_t *pid*, int * *wstatus*, int *options*, struct rusage * *rusage*);

η οποία ανακτά και επιστρέφει τον κωδικό επιστροφής της εξεταζόμενης διεργασίας, όπου:

pid → το process id της διεργασίας της οποίας αναμένεται ο κωδικός τερματισμού με τιμές

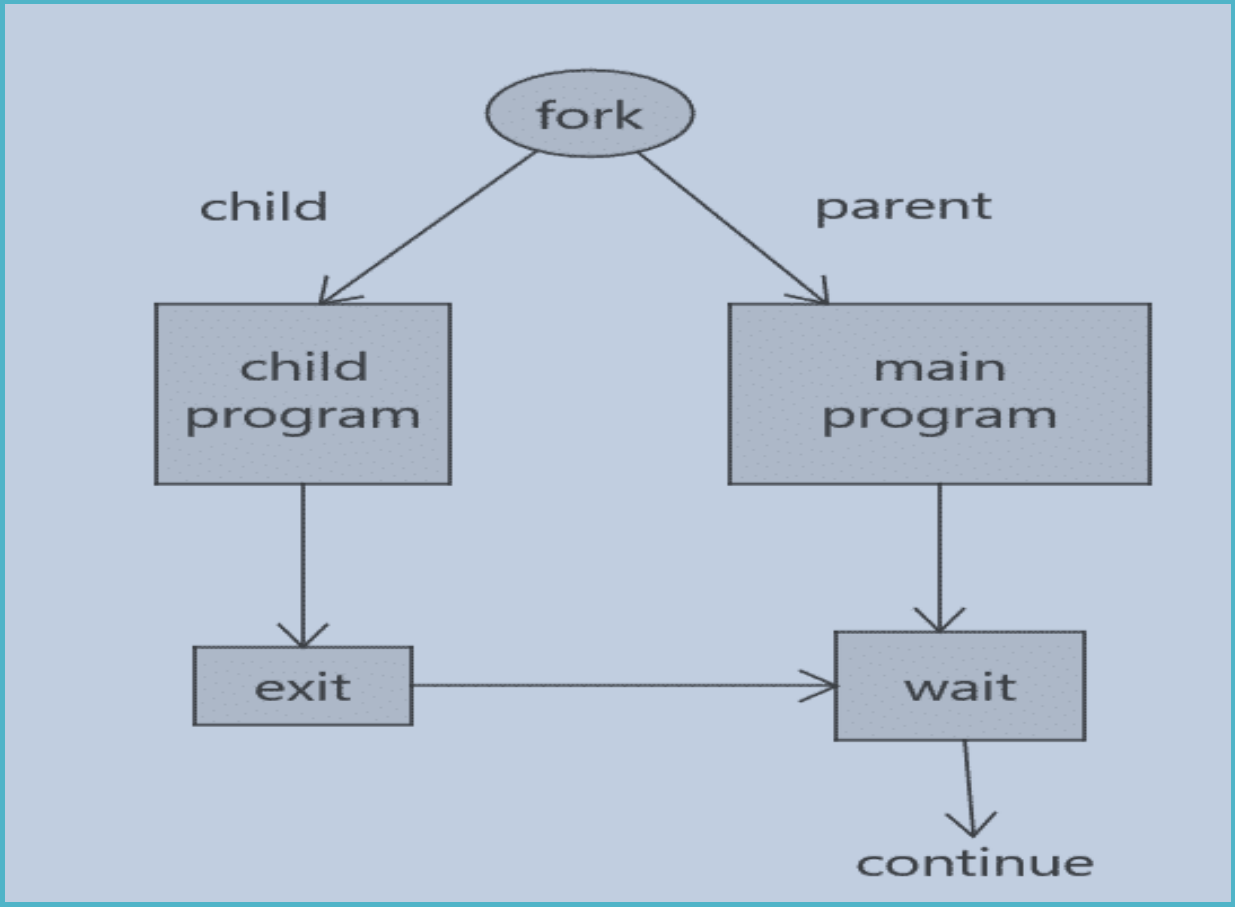
- < -1 → αναμένει τον τερματισμό οποιασδήποτε θυγατρικής διεργασίας της οποίας το pgid είναι το ίδιο με από απόλυτη τιμή της παραμέτρου pid.
- = -1 → αναμένει τον τερματισμό οποιασδήποτε θυγατρικής διεργασίας.
- = 0 → αναμένει τον τερματισμό διεργασιών που ανήκουν στην ίδια ομάδα διεργασιών με την τρέχουσα διεργασία.
- > 0 αναμένει τον τερματισμό της διεργασίας με process id ίσο με pid.

wstatus → δείκτης στην περιοχή αποθήκευσης του κωδικού τερματισμού της διεργασίας.

Κατά τη διάρκεια αυτής της διαδικασίας αναμονής του τερματισμού της θυγατρικής διεργασίας, η εκτέλεση της διεργασίας που κάλεσε τη συνάρτηση wait4, αναστέλλεται.

# Διαχείριση διεργασιών

Συνδυασμένη χρήση των fork, wait & exit



# Διαχείριση διεργασιών

## Εκτέλεση εντολών και προγραμμάτων μέσα από διεργασίες

Οι επόμενες οκτώ συναρτήσεις επιτρέπουν την εκτέλεση ενός προγράμματος μέσα από ένα άλλο και χαρακτηρίζονται από την ολική αντικατάσταση του προγράμματος που καλεί αυτές τις συναρτήσεις από το πρόγραμμα που καλείται.

Εάν επιθυμούμε τη διατήρηση του αρχικού προγράμματος θα πρέπει να δημιουργήσουμε μία νέα διεργασία καλώντας τη fork και στη συνέχεια να καλέσουμε την κατάλληλη από τις παραπάνω συναρτήσεις μέσα από τη νέα διεργασία.

```
int execl(const char *path, const char *arg0, ..., const char *argn, (char *)0);
int execle(const char *path, const char *arg0, ..., const char *argn, (char *)0, char *const envp[]);
int execlp(const char *file, const char *arg0, ..., const char *argn, (char *)0);
int execlpe(const char *file, const char *arg0, ..., const char *argn, (char *)0, char *const envp[]);
int execv(const char *path, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

Αυτές οι συναρτήσεις διαφέρουν απλά στον τρόπο με τον οποίο οι παράμετροι της εντολής κλήσης τους περνούν στο νέο πρόγραμμα.

# Διαχείριση διεργασιών

## Εκτέλεση εντολών και προγραμμάτων μέσα από διεργασίες

Οι παραπάνω συναρτήσεις διαχωρίζονται σε δύο κατηγορίες με κριτήριο το εάν το πέμπτο γράμμα του ονόματός τους είναι το **l** ή το **v**

execl execle execlp execlpe execv execve execvp execvpe

- Οι συναρτήσεις με το γράμμα l (list) δέχονται μία λίστα ορισμάτων (καταχωρημένα το ένα μετά το άλλο και χωρισμένα με κόμμα) η οποία τερματίζεται με NULL.
- **execl ("/bin/bash", "ls", "-l", "-R", "-a", NULL)**

- Οι συναρτήσεις με το γράμμα v (vector) δέχονται ως όρισμα έναν δείκτη προς ένα διάνυσμα δεικτών που περιέχει τα ορίσματα της συνάρτησης.

```
char * array [] = {"ls", "-l", "-R", "-a", NULL}
execl ("/bin/bash", array)
```

- Οι συναρτήσεις με το γράμμα p (path) αναζητούν το πρόγραμμα προς εκτέλεση στη λίστα των καταλόγων που βρίσκονται καταχωρημένοι στη μεταβλητή PATH.

- Οι συναρτήσεις με το γράμμα e (environment) δέχονται ένα δείκτη προς ένα νέο περιβάλλον προς το πρόγραμμα που φορτώνεται για εκτέλεση.

# Διαχείριση διεργασιών

## Οι συναρτήσεις system και popen

Εάν μία διεργασία καλέσει τη συνάρτηση system που ορίζεται ως

**int system (const char * cmd)**

περιμένει την ολοκλήρωση του προγράμματος cmd πριν συνεχίσει με την εκτέλεση του δικού της κώδικα. Η system επομένως δημιουργεί μία νέα θυγατρική διεργασία και εκτελεί το πρόγραμμα του φλοιού καλώντας την exec, το οποίο με τη σειρά του εκτελεί την εντολή cmd. Ο κωδικός επιστροφής της θυγατρικής διεργασίας διαβιβάζεται στη γονική διεργασία με τη wait.

**FILE * popen (const char * cmd, const char * mode)**

Η popen εκτελεί την εντολή cmd και δημιουργεί μία διασωλήνωση ανάμεσα στην εφαρμογή που κάλεσε την popen και στην εφαρμογή cmd επιστρέφοντας ένα δείκτη προς το ρεύμα δεδομένων που άνοιξε για διαδικασία ανάγνωσης ή εγγραφής.

Η cmd εκτελείται από το κέλυφος όπως και στη system ενώ το mode είναι “r” εάν η γονική διεργασία θέλει να διαβάσει την έξοδο της εντολής cmd και “w” εάν η εντολή cmd δέχεται είσοδο από τη διεργασία. Στο τέλος καλείται η

**int pclose (FILE * stream)**

που επιστρέφει τον κωδικό τερματισμού της θυγατρικής διεργασίας μέσω της wait.

# Διαχείριση διεργασιών

## Διαχείριση περιβάλλοντος

Η κάθε διεργασία που ξεκινά σε ένα σύστημα Linux με κάποιον από τους παραπάνω τρόπους έχει πρόσβαση στο σύνολο των μεταβλητών περιβάλλοντος του συστήματος.

Η ανάκτηση της τιμής μιας μεταβλητής περιβάλλοντος γίνεται με τη συνάρτηση

**char * getenv (const char * variable)**

Η προσθήκη περιεχομένου στο περιβάλλον γίνεται με τη συνάρτηση

**int putenv (char * string)**

όπου string συμβολοσειρά της μορφής variable = value ενώ η τροποποίηση γίνεται με τη συνάρτηση

**int setenv (const char * variable, const char * value, int overwrite)**

Τέλος η διαγραφή μιας μεταβλητής περιβάλλοντος γίνεται με τη συνάρτηση

**int unsetenv (const char * variable)**

Οι παραπάνω συναρτήσεις είναι δηλωμένες στο αρχείο επικεφαλίδας **stdlib.h**

# Διαχείριση διεργασιών

## Διαχείριση Πόρων

Για κάθε συνηθισμένη διεργασία που εκτελείται σε ένα σύστημα Linux υπάρχει άνω όριο σχετικά με τους πόρους (επεξεργαστή, μνήμη, δίσκο) που της διατίθενται για την εκτέλεσή της. Για κάθε πόρο ορίζονται δύο τιμές ορίων, μία για το μέγιστο όριο (hard) και μία για το ελάχιστο όριο (soft).

Η ανάκτηση των πόρων που έχουν ανατεθεί σε μία διεργασία γίνεται από τη συνάρτηση

**int getrlimit (int resource, struct rlimit * rlp)**

ενώ ο καθορισμός των πόρων γίνεται με τη βοήθεια της συνάρτησης

**int setrlimit (int resource, const struct rlimit * rlp)**

όπου rlimit κατάλληλα ορισμένη δομή δεδομένων που ορίζεται ως (<sys/resource.h>)

**struct rlimit { rlim_t rlim_cur; rlim_t rlim_max };**

Οι καθιερωμένοι πόροι είναι οι εξής:

- RLIMIT_CORE → μέγιστο μέγεθος αρχείου
- RLIMIT_CPU → μέγιστος χρόνος χρήσης της CPU (seconds)
- RLIMIT_DATA → μέγιστο μέγεθος τμήματος δεδομένων (bytes)
- RLIMIT_FSIZE → μέγιστο μέγεθος αρχείων (bytes)
- RLIMIT_NOFILE → μέγιστος αριθμός ταυτόχρονα ανοικτών αρχείων
- RLIMIT_STACK → μέγιστο μέγεθος στοίβας (bytes)

# Διαχείριση διεργασιών

## Η διεργασία init

Αποτελεί daemon process που ξεκινά κατά την εκκίνηση του υπολογιστή και εκτελείται συνεχώς μέχρι τον τερματισμό του που γίνεται με την εντολή

**shutdown –h [-r] now**

Είναι η γονική διεργασία όλων των διεργασιών που τρέχουν στο σύστημα και αναλαμβάνει τη διαχείριση των ορφανών διεργασιών. Η αποτυχημένη της εκκίνηση οδηγεί σε kernel panic.

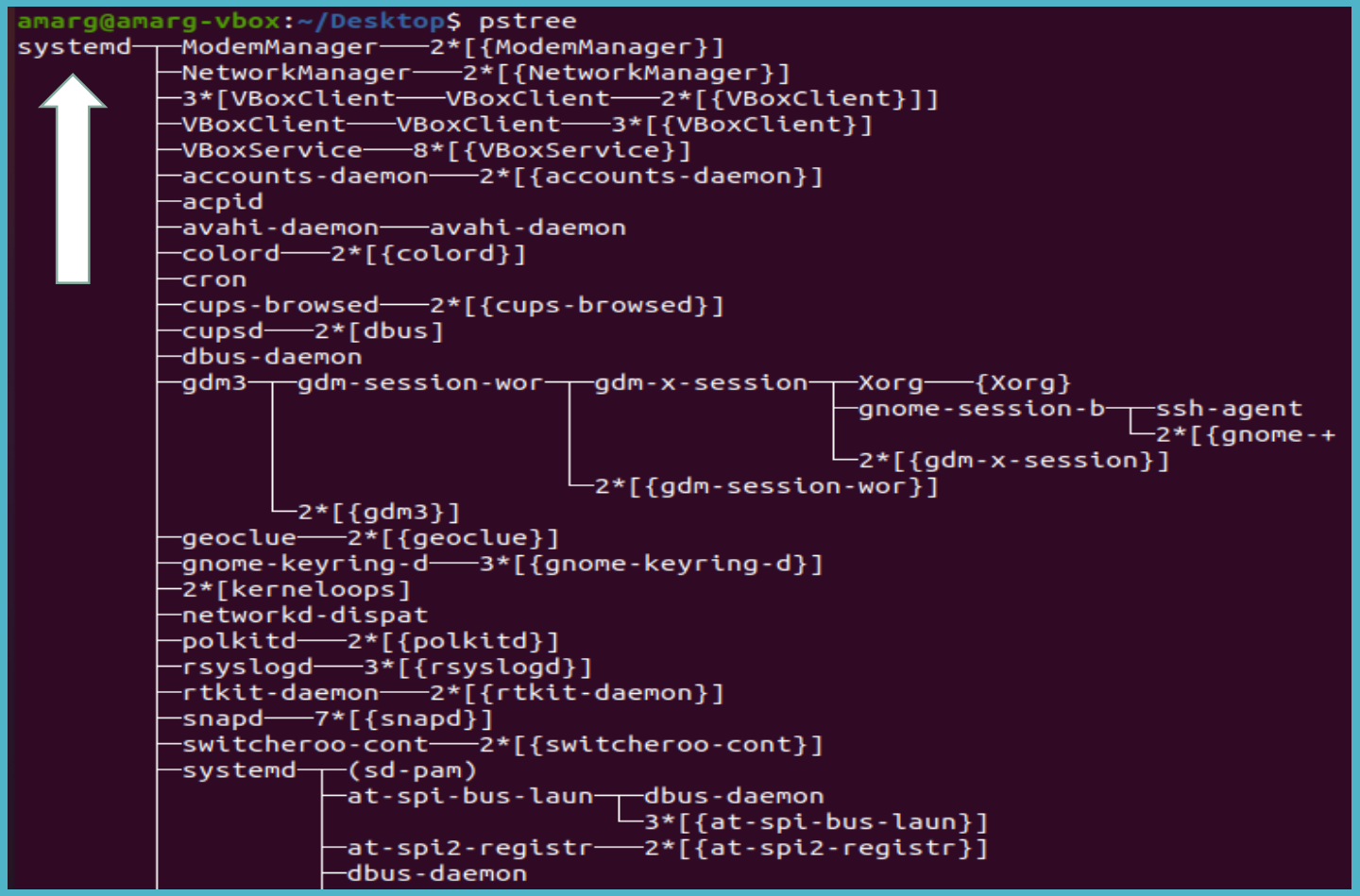
Η συμπεριφορά της καθορίζεται από τα περιεχόμενα του αρχείου /etc/inittab και μία από τις πιο σημαντικές παραμέτρους που ορίζονται εδώ είναι ο καθορισμός του runlevel εκκίνησης του συστήματος → **id:3:initdefault**

| | |
|---|---|
| **runlevel 0** | προκαλεί τον τερματισμό της λειτουργίας του συστήματος |
| **runlevel 1** | εκκινεί το λειτουργικό σύστημα σε κατάσταση απλού χρήστη **(single user mode)** |
| **runlevel 2** | εκκινεί το λειτουργικό σύστημα σε κατάσταση πολλών χρηστών **(multi user mode)** αλλά χωρίς υποστήριξη δικτύου |
| **runlevel 3** | εκκινεί το λειτουργικό σύστημα σε κατάσταση πολλών χρηστών **(multi user mode)** με ενεργοποιημένη την υποστήριξη δικτύου |
| **runlevel 4** | είναι δεσμευμένο αλλά δεν χρησιμοποιείται |
| **runlevel 5** | εκκινεί το σύστημα με ταυτόχρονη ενεργοποίηση του γραφικού περιβάλλοντος των **X Windows** |
| **runlevel 6** | προκαλεί την επανεκκίνηση του συστήματος. |

# Διαχείριση διεργασιών

## Η διεργασία systemd

Στις τελευταίες εκδόσεις του λειτουργικού συστήματος η διεργασία init έχει αντικατασταθεί από τη διεργασία systemd η οποία οδηγεί σε καλύτερη διαχείριση του συστήματος.



# Διαχείριση διεργασιών

## Μεταβολή προτεραιότητας διεργασίας – η εντολή nice

Η εντολή nice επιτρέπει τη μεταβολή της τιμής προτεραιότητας των διεργασιών. Η προεπιλεγμένη τιμή προτεραιότητας είναι η τιμή 0, η μέγιστη τιμή είναι -20 και η ελάχιστη είναι η 19.

Όσο μεγαλύτερη είναι η τιμή της προτεραιότητας μιας διεργασίας τόσο πιο μεγάλος είναι ο χρόνος που απασχολεί τη CPU και τόσο μεγαλύτερη είναι η εκχώρηση πόρων.

Η κλήση της nice χωρίς ορίσματα εκτυπώνει την τρέχουσα τιμή προτεραιότητας (συνήθως 0) ενώ η κλήση της εντολής με τη μορφή

**nice –nN cmd**

προκαλεί αύξηση της τιμής της προτεραιότητας της διεργασίας που προκύπτει από την εκτέλεση της εντολής cmd κατά N. Για παράδειγμα η εντολή

**nice –n10 gedit sample.txt**

ξεκινά την εφαρμογή gedit με τιμή προτεραιότητας 0 (default) + 10 = 10 καθιστώντας τη διεργασία που δημιουργείται λιγότερο επείγουσα σε σχέση με τις υπόλοιπες.

Σε επίπεδο κώδικα C η συνάρτηση nice ορίζεται ως **int nice (int increment)**.

# Διαχείριση διεργασιών

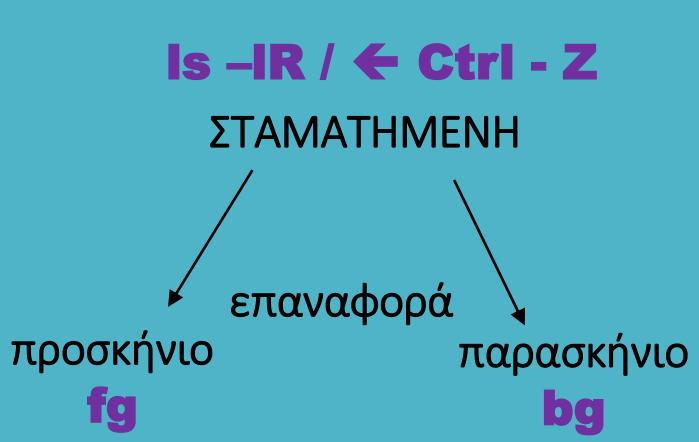
## Προσκήνιο και παρασκήνιο

Μία διεργασία εκτελείται στο προσκήνιο (foreground) όταν εκτελείται από τη γραμμή εντολών και κατά συνέπεια απασχολεί το φλοιό.

Μία διεργασία εκτελείται στο παρασκήνιο (background) όταν δεν δεσμεύει το φλοιό και ως εκ τούτου παράλληλα με την εκτέλεσή της μπορούμε μέσω του φλοιού να αλληλεπιδράσουμε με το σύστημα. Μια διεργασία μπορεί να ξεκινήσει απευθείας στο παρασκήνιο με τη βοήθεια του τελεστή & που καταχωρείται ως τελευταίος χαρακτήρας κατά την κλήση της δηλαδή ως

**cmd [arguments] &**

Εάν επιθυμούμε την αναστολή της διεργασίας που εκτελείται χρησιμοποιούμε το συνδυασμό Ctrl-Z.




```
amarg@amarg-vbox:~/Desktop$ ps -x | sort -k1 -r -n
  13839 pts/0    S+     0:00 sort -k1 -r -n
  13838 pts/0    R+     0:00 ps -x
  13831 pts/0    T      0:00 ls --color=auto -lR /
  13825 pts/0    Ss     0:00 bash
```

```
amarg@amarg-vbox:~/Desktop$ jobs
[1]+  Stopped                 ls --color=auto -lR /
```

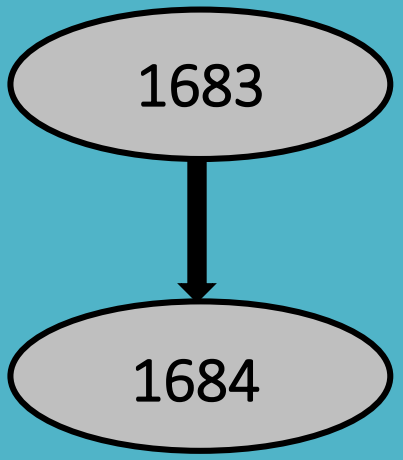
Το job ως έννοια σχετίζεται με το φλοιό

# Διαχείριση διεργασιών

## Παράδειγμα 1

Δημιουργία θυγατρικής διεργασίας με κλήση της fork().

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

void  main(void) {
     pid_t  pid;
     pid = fork();
     printf ("Hello !! my pid is %d\n", getpid()); }
```




Η printf εκτελείται και από τη γονική και από τη θυγατρική διεργασία (για αυτό και εκτυπώνονται δύο μηνύματα ενώ υπάρχει μία και μοναδική printf) και εκτυπώνει το pid της καθεμίας από αυτές.

```
amarg@amarg-vbox:~$ ./shared/Lab4/forkex1
Hello !! my pid is 1683
amarg@amarg-vbox:~$ Hello !! my pid is 1684
```

Η γονική διεργασία ολοκληρώθηκε πρώτη (αν και θα μπορούσε πρώτη να ολοκληρωθεί η θυγατρική) και για το λόγο αυτό το shell prompt εμφανίστηκε πριν την ολοκλήρωση της θυγατρικής διεργασίας.

# Διαχείριση διεργασιών

## Παράδειγμα 2

```c
#include <unistd.h>     /* for fork(), getpid() */
#include <sys/types.h>  /* for waitpid() */
#include <sys/wait.h>   /* for waitpid() */
#include <stdlib.h>     /* for exit() */
#include <stdio.h>      /* for printf(), perror() */

int main(int argc, char *argv[]) {
    pid_t child_pid = fork();
    if (child_pid < 0) {
        perror("fork() failed");
        exit(EXIT_FAILURE);
    } else if (child_pid == 0) {
        printf("from child: pid=%d, parent_pid=%d\n", (int) getpid(), (int) getppid());
        exit(33);
    } else if (child_pid > 0) {
        printf("from parent: pid=%d child_pid=%d\n", (int) getpid(), (int) child_pid);
        int status;
        pid_t waited_pid = waitpid(child_pid, &status, 0);
        if (waited_pid < 0) {
            perror("waitpid() failed");
            exit(EXIT_FAILURE);
        } else if (waited_pid == child_pid) {
            if (WIFEXITED(status)) {
                printf("from parent: child exited with code %d\n", WEXITSTATUS(status)); }}}
    exit(EXIT_SUCCESS); }
```

OUTPUT

```
amarg@amarg-vbox:~$ ./shared/Lab4/forkex2
from parent: pid=1804 child_pid=1805
from child: pid=1805, parent_pid=1804
from parent: child exited with code 33
```

Ο γονέας περιμένει τον τερματισμό του παιδιού

# Διαχείριση διεργασιών

## Παράδειγμα 3

```
int global = 0;
int main() {
    pid_t child_pid;
    int status;
    int local = 0;
    child_pid = fork();

    if (child_pid >= 0) /* fork succeeded */ {

        if (child_pid == 0) /* fork() returns 0 for the child process */ {
            printf("child process!\n");
            local++; global++;
            printf("child PID =  %d, parent pid = %d\n", getpid(), getppid());
            printf("\nchild's local = %d, child's global = %d\n",local,global);
            char * cmd[] = {"whoami",(char*)0};
            return execv("/usr/bin/whoami",cmd); /* call whoami command */ }

         else /* parent process */ {
             printf("parent process!\n");
             printf("parent PID =  %d, child pid = %d\n", getpid(), child_pid);
             wait(&status); /* wait for child to exit, and store child's exit status */
             printf("Child exit code: %d\n", WEXITSTATUS(status));
             /* The change in local and global variable in child process */
             /* should not reflect here in parent process. */
             printf("\nParent'z local = %d, parent's  global = %d\n",local,global);
             printf("Parent says bye!\n");
             exit(0);  /* parent exits */ }}

    else /* failure */ {
        perror("fork");
        exit(0); }}
```

```
parent process!
parent PID =  1843, child pid = 1844
child process!
child PID =  1844, parent pid = 1843

child's local = 1, child's global = 1
amarg
Child exit code: 0

Parent'z local = 0, parent's  global = 0
Parent says bye!
```

OUTPUT

Οι μεταβλητές local και global έχουν διαφορετικές τιμές στις δύο διεργασίες !

# Διαχείριση διεργασιών

## Η δομή του κώδικα χρήσης της fork

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

int main() {
    pid_t pid, pid1;

A   Κώδικας πριν την κλήση της fork.
    Υπάρχει μία και μοναδική διεργασία
    η οποία είναι η γονική διεργασία

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;}

    else if (pid == 0) { /* child process */

         Γ  Κώδικας που εκτελείται από
            τη θυγατρική διεργασία
    }
    else { /* parent process */

         B  Κώδικας που εκτελείται
            από τη γονική διεργασία
    }

    return 0;}
```

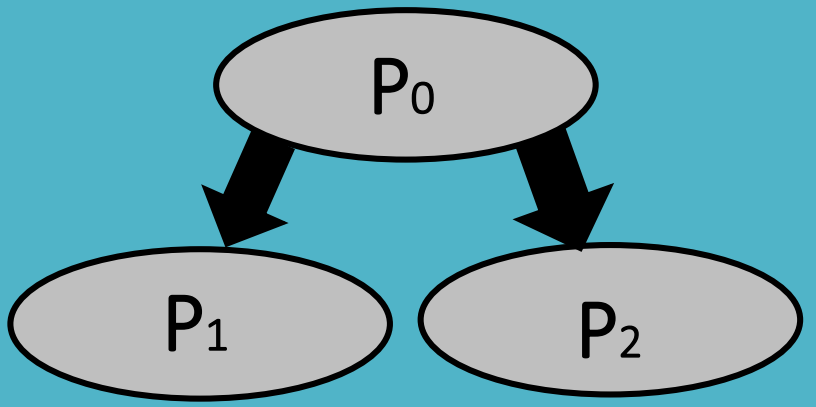
Εάν λοιπόν θέλουμε να γράψουμε κώδικα που να εκτελείται από τη γονική διεργασία, αυτός μπορεί να γραφεί είτε στην Ενότητα Α είτε στην Ενότητα Β, οι οποίες είναι ισοδύναμες.

Αντίθετα ο κώδικας για τη θυγατρική διεργασία γράφεται ΜΟΝΟ στην Ενότητα Γ

# Διαχείριση διεργασιών

## Τοπολογίες διεργασιών

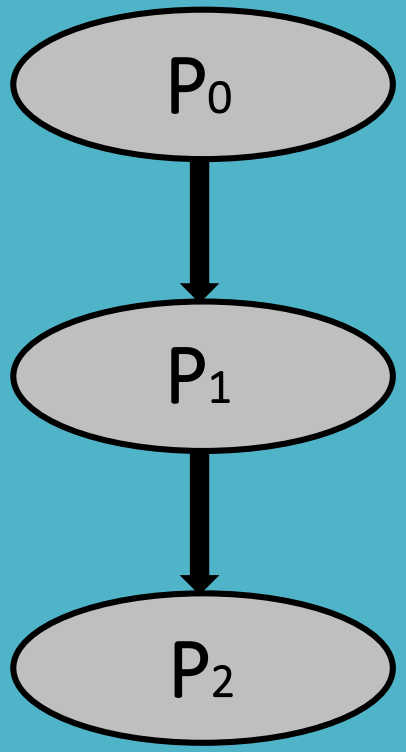
```
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>

int main () {
    int pid1, pid2;
    pid1 = fork();
    if (pid1>0) {
                        Κώδικας για τη
                        διεργασία P0

        pid2 = fork();
        if (pid2>0)
            {
                        Κώδικας για τη
                        διεργασία P0
            }
         else {
                        Κώδικας για τη
                        διεργασία P2
              }
     }
    else {
                        Κώδικας για τη
                        διεργασία P1
          }}
```



# Διαχείριση διεργασιών

## Τοπολογίες διεργασιών

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

int main () {
    int pid1, pid2;
    pid1 = fork();
    if (pid1>0) {
                        Κώδικας για τη
                        διεργασία P0
                }
    else {
                                    Κώδικας για τη
                                    διεργασία P1
            pid2 = fork();
            if (pid2>0)
              {
                                    Κώδικας για τη
                                    διεργασία P1
              }
            else {
                                    Κώδικας για τη
                                    διεργασία P2
                }

                    Κώδικας για τη
                    διεργασία P1
        }}
```



# Διαχείριση διεργασιών

## Παράδειγμα 4

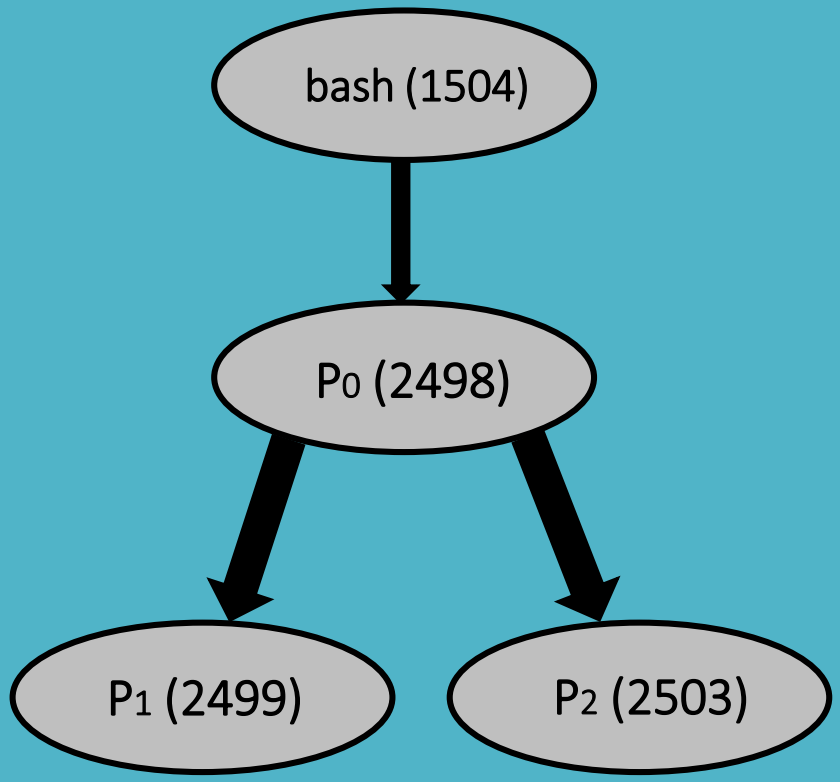
```
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>

int main () {
    int pid1, pid2, status1, status2, child1, child2;
    pid1 = fork();
    if (pid1<0) {
        printf ("Fork operation was uncussesfull\n");
        return -1 ; }
    else if (pid1>0) {
          printf ("Parent process with pid %d and ppid %d \n",getpid(), getppid());
          child1 = wait(&status1);
          printf ("Child with code %d has beem terminated\n", child1);
          pid2 = fork();
          if (pid2>0)
              {
                printf ("Parent process \n");
                child2 = wait(&status2);
                printf ("Child with code %d has been terminated\n", child2); }
           else {
                printf ("Child2 with pid %d and ppid %d \n", getpid(), getppid());
                printf ("Calling pwd command...");
                execl ("/usr/bin/pwd","pwd",NULL); }}
    else {
            printf ("Child1 with pid %d kai ppid %d \n", getpid(), getppid());
            printf ("Creating a new directory...\n");
            execlp ("mkdir", "mkdir", "OSLab", NULL); }}
```

# Διαχείριση διεργασιών

## Παράδειγμα 4

```
amarg@amarg-vbox:~$ shared/Lab4/forkex4
Parent process with pid 2498 and ppid 1504
Child1 with pid 2499 kai ppid 2498
Creating a new directory...
Child with code 2499 has beem terminated
Parent process
Child2 with pid 2503 and ppid 2498
/home/amarg
Child with code 2503 has been terminated
```

```
amarg@amarg-vbox:~$ ps
    PID TTY          TIME CMD
   1504 pts/0    00:00:00 bash
   2504 pts/0    00:00:00 ps
```



# Διαχείριση διεργασιών

## Παράδειγμα 5

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

int main () {
    int pid1, pid2, status1, status2, child1, child2;
    pid1 = fork();
    if (pid1<0) {
        printf ("Fork operation was uncussesfull\n");
        return -1 ; }
    else if (pid1>0) {
          printf ("Parent process with pid %d and ppid %d \n",getpid(), getppid());
          child1 = wait(&status1);
          printf ("Child with code %d has beem terminated\n", child1);
          }
    else {
            printf ("Child1 with pid %d kai ppid %d \n", getpid(), getppid());
            pid2 = fork();
            if (pid2>0)
              {
                printf ("Parent process (child1) for child2\n");
                child2 = wait(&status2);
                printf ("Child with code %d has been terminated\n", child2); }
           else {
                printf ("Child2 with pid %d and ppid %d \n", getpid(), getppid());
                printf ("Calling pwd command...");
                execl ("/usr/bin/pwd","pwd",NULL); }

            printf ("Creating a new directory...\n");
            execlp ("mkdir", "mkdir", "OSLab", NULL); }}
```

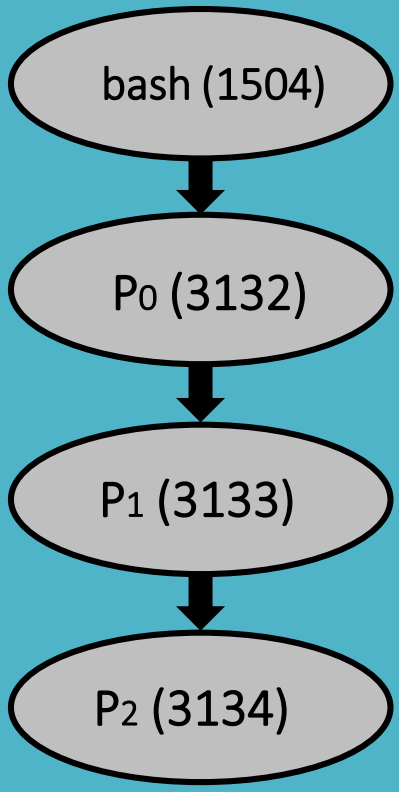
# Διαχείριση διεργασιών

## Παράδειγμα 5

```
amarg@amarg-vbox:~$ ./shared/lab4/forkex5
Parent process with pid 3132 and ppid 1504
Child1 with pid 3133 kai ppid 3132
Parent process (child1) for child2
Child2 with pid 3134 and ppid 3133
/home/amarg
Child with code 3134 has been terminated
Creating a new directory...
mkdir: cannot create directory ‘OSLab’: File exists
Child with code 3133 has beem terminated
```

```
 1496 ?        Ssl    0:14 /usr/libexec/gnome-terminal
 1504 pts/0    Ss     0:00 bash
 1541 ?        Ssl    0:00 /usr/libexec/gvfsd-metadata
 1544 ?        Sl     0:01 update-notifier
 2627 ?        Sl     0:00 /usr/libexec/gvfsd-network
 2646 ?        Sl     0:00 /usr/libexec/gvfsd-dnssd --
 2759 pts/1    Ss     0:00 bash
 2958 pts/2    Ss+    0:00 bash
 3144 ?        Ssl    0:00 /usr/libexec/tracker-store
 3151 pts/0    S+     0:00 ./forkex6
 3152 pts/0    S+     0:00 ./forkex6
 3153 pts/0    S+     0:00 ./forkex6
```

Διαφορετικά pids για διαφορετικές εκτελέσεις




Έξοδος της ps -xelf

```
0  1000    1496     861  20   0 826352 54472 -      Ssl  ?        0:14 \_ /usr/libexec/gnome-terminal-server HOME=/h
0  1000    1504    1496  20   0  19644  5388 do_wai Ss   pts/0    0:00 |   \_ bash GJS_DEBUG_TOPICS=JS ERROR;JS LOG
0  1000    3151    1504  20   0   2488   580 do_wai S+   pts/0    0:00 |   |   \_ ./forkex6 SHELL=/bin/bash SESSION_
1  1000    3152    3151  20   0   2488    76 do_wai S+   pts/0    0:00 |   |       \_ ./forkex6 SHELL=/bin/bash SESS
1  1000    3153    3152  20   0   2488    84 hrtime S+   pts/0    0:00 |   |           \_ ./forkex6 SHELL=/bin/bash
```