Διαδικασία (process)

- Στιγμιότυπο ενός προγράμματος που εκτελείται
- Λειτουργικό φορτώνει πρόγραμμα στη μνήμη
- Ανάθεση αναγνωριστικού στη διεργασία

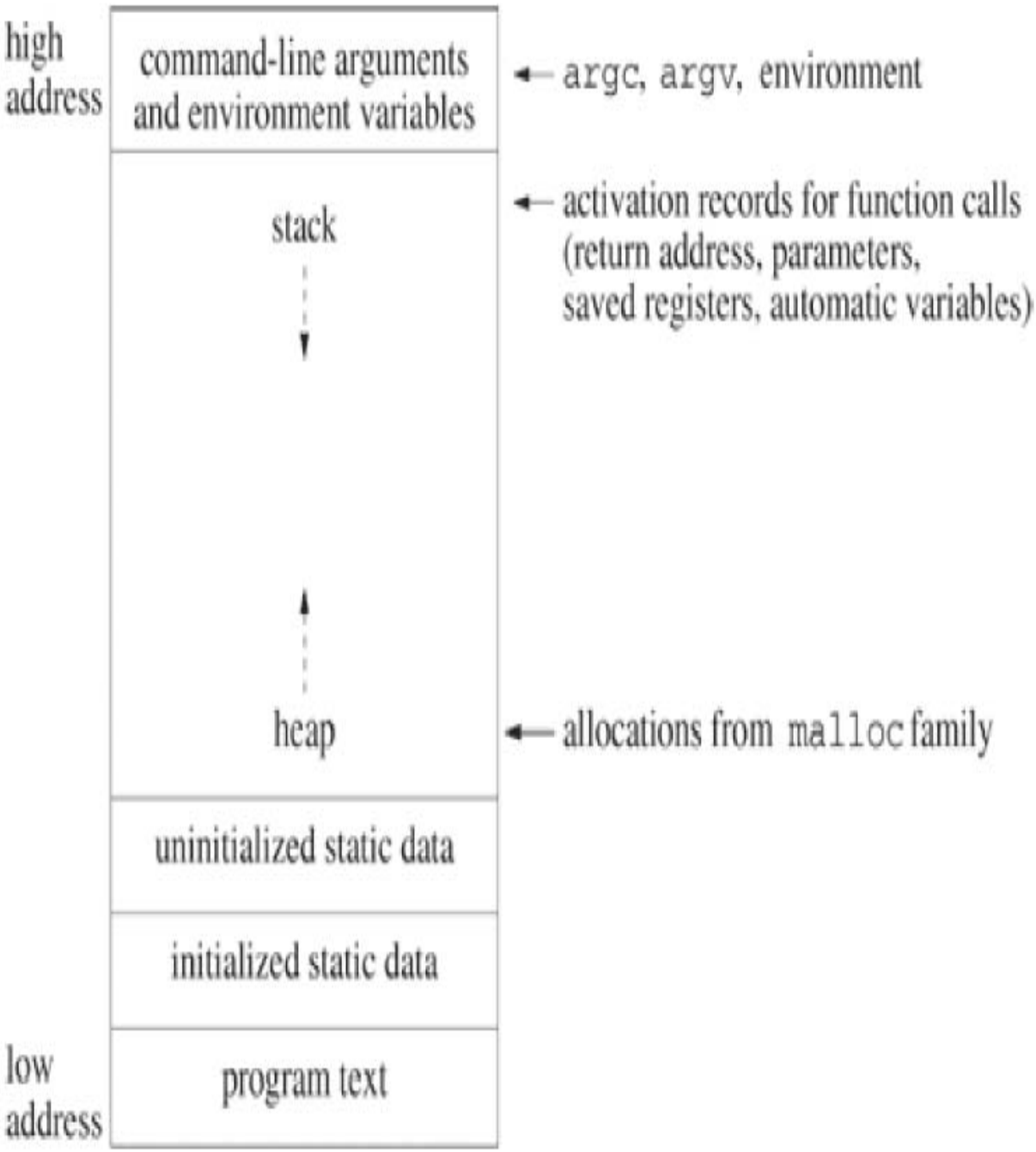
- Κάθε διαδικασία έχει
  - Χώρος διευθύνσεων
    - Εντολές προγράμματος
    - Αρχικοποιημένα και μη αρχικοποιημένα δεδομένα
    - Στοίβα εκτέλεσης

Διαδικασία (συν.)

- Ξεκινάει με μοναδική σειρά εκτέλεσης εντολών
- Μετρητής προγράμματος του επεξεργαστή: Κρατάει ποια εντολή θα εκτελεστεί μετά στο συγκεκριμένο επεξεργαστή

- Διαδικασίες επικοινωνούν μέσω συστήματος αρχείων, σωλήνες, κοινή μνήμη, δικτύου…

Εικόνα Προγράμματος



Περι διεργασιών…

- Κάθε διεργασία στο Unix έχει έναν αριθμό ταυτότητας (PID), τον κώδικά της, τα δεδομένα της, μία στοίβα καθώς και κάποια άλλα χαρακτηριστικά
- Η αρχική διεργασία είναι η init (PID=1)
- Ο μόνος τρόπος να δημιουργηθεί μία διεργασία είναι κάποια άλλη να αναπαραγάγει τον εαυτό της, δηλαδή μία διεργασία-γονέας να γεννήσει μία διεργασία-παιδί
- Όλες οι διεργασίες είναι απόγονοι της init
- Ο κώδικας, τα δεδομένα και η στοίβα της διεργασίας-παιδιού είναι αντίγραφα αυτών της διεργασίας-γονέα
- Η ταυτότητα της διεργασίας-παιδιού είναι διαφορετική από την ταυτότητα της διεργασίας-γονέα
- Μία διεργασία-παιδί μπορεί να αντικαταστήσει τον κώδικα, τα δεδομένα και τη στοίβα της με αυτά ενός εκτελέσιμου αρχείου, διαφοροποιώντας έτσι τον εαυτό της από το γονέα της

Όρια διεργασιών…

- Συναρτήσεις getlimit, setlimit

```
struct rlimit {
    rlimit_t rlim_cur;   //soft limit = current limit
    rlimit_t rlim_max;  //hard limit = max value
                        // for current limit
}
```

< sys/time.h>
< sys/resource.h>
int getrlimit(int resource, struct rlimit *reslimp);
    Επιστρέφει 0, αν OK, ενώ σε σφάλμα επιστρέφει <> 0
int setrlimit(int resource, const struct rlimit *reslimp);
    Επιστρέφει 0, αν OK, ενώ σε σφάλμα επιστρέφει <> 0

Πρόγραμμα testlimits

```
#include <sys/time.h>
#include <sys/resource.h>
#include <stdio.h>

int main() {
  struct rlimit rlimit;

  getrlimit(RLIMIT_AS, &rlimit);
  printf("Maximum address space = %lu and current = %lu\n",
         rlimit.rlim_max, rlimit.rlim_cur);

  getrlimit(RLIMIT_CORE, &rlimit);
  printf("Maximum core file size = %lu and current = %lu\n",
         rlimit.rlim_max, rlimit.rlim_cur);

  getrlimit(RLIMIT_DATA, &rlimit);
  printf("Maximum data+heap size = %lu and current = %lu\n",
         rlimit.rlim_max, rlimit.rlim_cur);

  getrlimit(RLIMIT_FSIZE, &rlimit);
  printf("Maximum file size = %lu and current = %lu\n",
         rlimit.rlim_max, rlimit.rlim_cur);

  getrlimit(RLIMIT_STACK, &rlimit);
  printf("Maximum stack size = %lu and current = %lu\n",
         rlimit.rlim_max, rlimit.rlim_cur);

  return 1;
}
```

Εκτέλεση testlimits

```
linux03:/home/users/spro>./testlimits
Maximum address space = 4294967295 and current = 4294967295
Maximum core file size = 4294967295 and current = 0
Maximum data+heap size = 4294967295 and current = 4294967295
Maximum file size = 4294967295 and current = 4294967295
Maximum stack size = 4294967295 and current = 8388608
linux03:/home/users/spro>
```

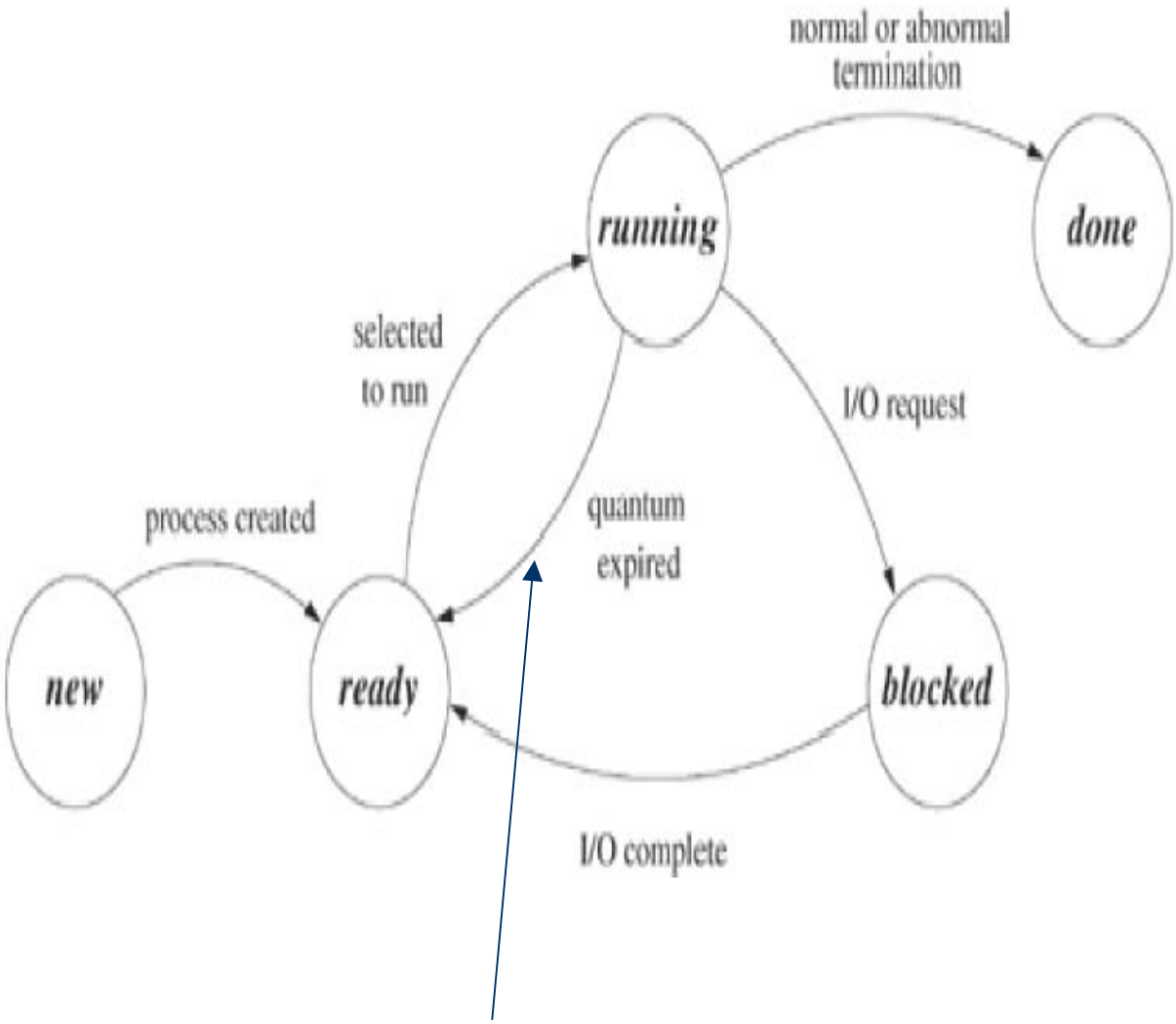
Αναγνώριση Διεργασιών

```
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void) ;

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

int main (void) {
   printf("I am process %ld\n", (long)getpid());
   printf("My parent is %ld\n", (long)getppid());
   return 0;
}
```

Αναγνωριστικό δικό μου

Αναγνωριστικό του πατέρα μου

Κατάσταση Διεργασίας



Εδώ αποφασίζεται να εκτελεστεί κάποια άλλη διεργασία (context switching)

Κλήση συστήματος exit

- Κλήση συστήματος `exit`
  - `void exit(int status)`
  - Τερματίζει την εκτέλεση μίας διεργασίας
  - Ο ακέραιος `status` ονομάζεται κωδικός εξόδου και είναι διαθέσιμος στη διεργασία-γονέα
  - Κατά σύμβαση, η τιμή 0 για το `status` δείχνει επιτυχή τερματισμό
  - Πρόσβαση με το `$status` στο κέλυφος C και με το `$?` στο κέλυφος Bourne
  - Απαίτηση: `#include <stdlib.h>`
- Χρήση της κλήσης `exit`

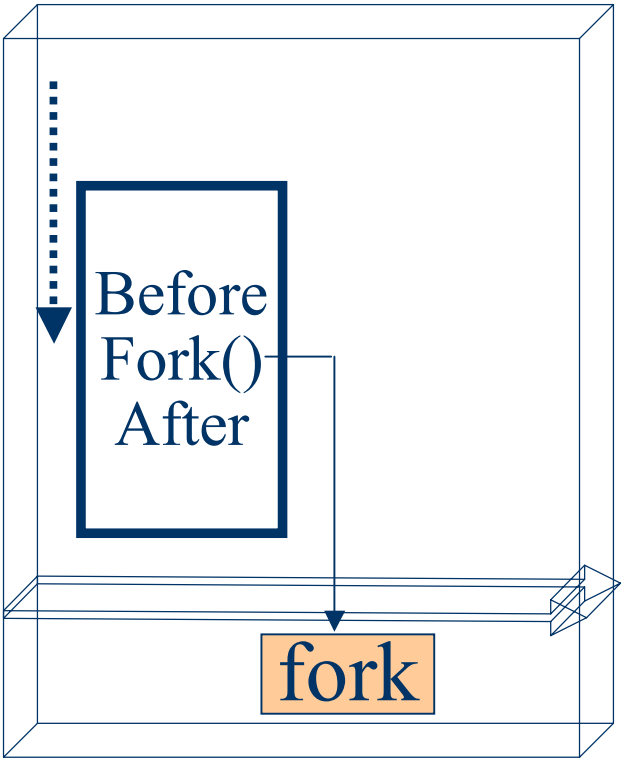
```
/* File: exit_code.c */
#include <stdio.h>   /* For printf */
#include <stdlib.h>    /* For exit */
main()
{  printf("I am going to terminate with exit code 23\n");
   exit(23); }
```

```
$ ./exit_code
I am going to terminate with exit code 23
$ echo $?
23
$
```

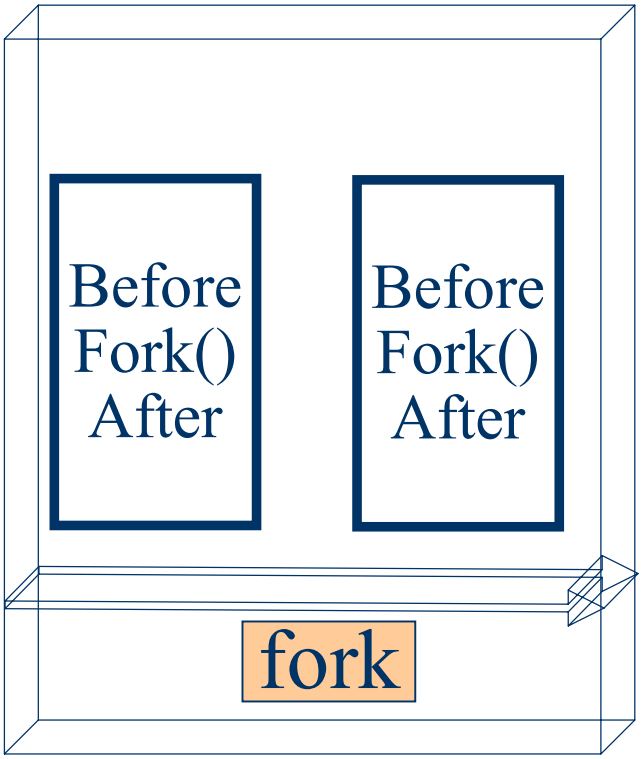
Δημιουργία διεργασίας - fork

- `int fork()`
- Αναπαράγει μία διεργασία, δημιουργώντας μία άλλη πανομοιότυπη
- Στη διεργασία-γονέα επιστρέφει την ταυτότητα της διεργασίας-παιδιού και στη διεργασία-παιδί επιστρέφει 0
- Επιστρέφει λάθος (-1) στη διεργασία-γονέα αν δεν είναι δυνατόν να δημιουργηθεί η νέα διεργασία-παιδί

Πριν το fork



Μετά το fork



Παράδειγμα fork

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void) {
  pid_t childpid;

  childpid = fork();
  if (childpid == -1) {
    perror("Failed to fork");
    return 1;
  }
  if (childpid == 0)                    /* child code */
    printf("I am child %ld\n",  (long)getpid());
  else                                  /* parent code */
    printf("I am parent %ld\n",  (long)getpid());
  return 0;
}
```

Στο παιδί επιστρέφει 0, στον πατέρα το ID του παιδιού

Παράδειγμα fork (2)

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

int main(void) {
   pid_t childpid;
   pid_t mypid;

   mypid = getpid();
   childpid = fork();
   if (childpid == -1) {
      perror("Failed to fork");
      return 1;
   }
   if (childpid == 0)                    /* child code */
      printf("I am child %ld, ID = %ld\n", (long)getpid(), (long)mypid);
   else                                  /* parent code */
      printf("I am parent %ld, ID = %ld\n", (long)getpid(), (long)mypid);
   return 0;
}
```

ID του πατέρα. Το mypid αντιγράφεται στο παιδί μετά το fork

getpid() <> mypid

getpid() == mypid

Αλυσίδα διαδικασιών

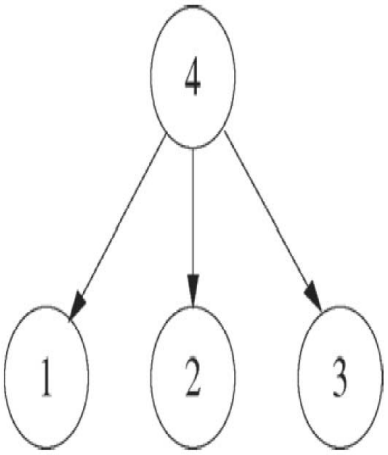
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char *argv[]) {
   pid_t childpid = 0;
   int i, n;

   if (argc != 2){   /* check for valid number of command-line arguments */
      fprintf(stderr, "Usage: %s processes\n", argv[0]);
      return 1;
   }
   n = atoi(argv[1]);
   for (i = 1; i < n; i++)
      if (childpid = fork())
         break;

   fprintf(stderr, "i:%d  process ID:%ld  parent ID:%ld  child ID:%ld\n",
         i, (long)getpid(), (long)getppid(), (long)childpid);
   return 0;
}
```

Μόνο το παιδί κάθε fork συνεχίζει το loop

Δε θα τυπωθούν στη σειρά. Εξαρτάται από δρομολογητή διεργασιών

Ρηχό δέντρο διαδικασιών

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

int main (int argc, char *argv[]) {
  pid_t childpid = 0;
  int i, n;

  if (argc != 2){   /* check for valid number of command-line arguments */
    fprintf(stderr, "Usage: %s processes\n", argv[0]);
    return 1;
  }
  n = atoi(argv[1]);
  for (i = 1; i < n; i++)
    if ((childpid = fork()) <= 0)
      break;
  fprintf(stderr, "i:%d  process ID:%ld  parent ID:%ld  child ID:%ld\n",
          i, (long)getpid(), (long)getppid(), (long)childpid);
  return 0;
}
```



Μόνο ο πατέρας κάθε fork συνεχίζει το loop

Τι θα γίνει αν βάλω:

if (childpid = fork()) == -1) ???

Ορφανές διαδικασίες

- Αν μία διεργασία τερματίσει πρίν από κάποιο παιδί της, τότε το τελευταίο, σαν ορφανή (orphan) διεργασία, υιοθετείται από την init

```
/* File: orphan_process.c */
#include <stdio.h>                              /* For printf */
#include <stdlib.h>                              /* For exit */
main()
{  int pid;
   printf("Original process: PID = %d, PPID = %d\n",
          getpid(), getppid());
   pid = fork();
   if (pid == -1) {                      /* Check for error */
      perror("fork");
      exit(1); }
   if (pid != 0) {                     /* The parent process */
      printf("Parent process: PID = %d, PPID = %d, CPID = %d\n",
             getpid(), getppid(), pid); }
   else {                                /* The child process */
      sleep(2);           /* Delay child in order to ensure
                             that the parent terminates first */
      printf("Child process: PID = %d, PPID = %d\n",
             getpid(), getppid()); }
   printf("Process with PID = %d terminates\n",
          getpid()); }        /* Executed by both processes */
```

```
$ ./orphan_process
Original process: PID = 1587, PPID = 1541
Parent process: PID = 1587, PPID = 1541, CPID = 1588
Process with PID = 1587 terminates
$ Child process: PID = 1588, PPID = 1
Process with PID = 1588 terminates
```

Κλήση συστήματος wait

- `int wait(int *status)`
- Προκαλεί την αναμονή μίας διεργασίας μέχρις ότου κάποιο παιδί της τερματίσει
- Αποδέχεται τον κωδικό εξόδου του παιδιού, δηλαδή τον αριθμό της κλήσης `exit`, στο αριστερό byte του `status`, ενώ το δεξιό byte είναι 0
- Αν το παιδί τερματίσει εξ αιτίας κάποιου σήματος, τότε τα 7 τελευταία bits του `status` αντιπροσωπεύουν το χαρακτηριστικό αριθμό αυτού του σήματος
- Επιστρέφει την ταυτότητα του παιδιού που τερμάτισε ή λάθος (-1) αν η διεργασία δεν έχει παιδιά

ΠΑΝΤΑ Ο ΠΑΤΕΡΑΣ ΝΑ ΚΑΛΕΙ ΤΗΝ WAIT ΓΙΑ ΚΑΘΕ ΠΑΙΔΙ

Αποδεσμεύονται πόροι διεργασίας

Η init περιοδικά αποδεσμεύει χώρο διεργασιών που δεν τις περίμενε ο πατέρας τους

Κλήση waitpid

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

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

Αν pid = -1, η waitpid περιμένει για οποιοδήποτε παιδί.

Αν options = WNOHANG, επιστρέφει αμέσως 0 αν υπάρχουν παιδιά για τα οποία δε γνωρίζει το status.

```
pid_t childpid;

while (childpid = waitpid(-1, NULL, WNOHANG))
   if ((childpid == -1) && (errno != EINTR))
      break;
```

Επανεκκίνηση αν διακόπηκε από σήμα

Παραδείγματα χρήσης wait

```
/* File: wait_usage.c */
#include <stdio.h>                                  /* For printf */
#include <stdlib.h>                                   /* For exit */
main()
{  int pid, status;
   printf("Original process: PID = %d\n", getpid());
   pid = fork();
   if (pid == -1) {                              /* Check for error */
      perror("fork");
      exit(1); }
   if (pid != 0) {                             /* The parent process */
      printf("Parent process: PID = %d\n", getpid());
      if (wait(&status) != pid) {  /* Wait for child to exit */
         perror("wait");
         exit(1); }
      printf("Child terminated: PID = %d, exit code = %d\n",
            pid, status >> 8); }
   else {                                        /* The child process */
      printf("Child process: PID = %d, PPID = %d\n",
            getpid(), getppid());
      exit(72); }                          /* Exit with a silly number */
   printf("Process with PID = %d terminates\n",
         getpid()); }                      /* Executed by parent only */
```

```
$ ./wait_usage
Original process: PID = 1591
Child process: PID = 1592, PPID = 1591
Parent process: PID = 1591
Child terminated: PID = 1592, exit code = 72
Process with PID = 1591 terminates
$
```

Ζωντανές-νεκρές διεργασίες

- Μία διεργασία που τερματίζει δεν εγκαταλείπει το σύστημα μέχρις ότου ο γονέας της αποδεχθεί τον κωδικό εξόδου της και είναι όλο αυτό το χρονικό διάστημα μία ζωντανή-νεκρή (zombie) διεργασία

```
/* File: zombie_process.c */
#include <stdlib.h>                    /* For exit */
main()
{  int pid;
   pid = fork();
   if (pid == -1) {              /* Check for error */
      perror("fork");
      exit(1); }
   if (pid != 0) {            /* The parent process */
      while (1)                  /* Never terminate */
         sleep(1000); }
   else {                      /* The child process */
      exit(37); } }    /* Exit with a silly number */
```

```
$ ./zombie_process &
[1] 1593
$ ps -a
  PID TTY          TIME CMD
 1593 pts/0    00:00:00 zombie_process
 1594 pts/0    00:00:00 zombie_process <defunct>
 1597 pts/0    00:00:00 ps
$ kill 1593
$
[1]+  Terminated              ./zombie_process
$ ps -a
  PID TTY          TIME CMD
 1600 pts/0    00:00:00 ps
$
```

Τι μπορεί να τυπωθεί εδώ?

Σκεφτείτε πότε μπορεί να έρθει ένα σήμα

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

int main (void) {
  pid_t childpid;
                    /* set up signal handlers here ... */
  childpid = fork();
  if (childpid == -1) {
    perror("Failed to fork");
    return 1;
  }
  if (childpid == 0)
    fprintf(stderr, "I am child %ld\n", (long)getpid());
  else if (wait(NULL) != childpid)
    fprintf(stderr, "A signal must have interrupted the wait!\n");
  else
    fprintf(stderr, "I am parent %ld with child %ld\n", (long)getpid(),
        (long)childpid);
  return 0;
}
```

Τι τυπώνεται εδώ?

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main (int argc, char *argv[]) {
  pid_t childpid = 0;
  int i, n;

  if (argc != 2){      /* check number of command-line arguments */
    fprintf(stderr, "Usage: %s processes\n", argv[0]);
    return 1;
  }
  n = atoi(argv[1]);
  for (i = 1; i < n; i++)
    if ((childpid = fork()) <= 0)
      break;
  for( ; ; ) {
    childpid = wait(NULL);
    if ((childpid == -1) && (errno != EINTR))
      break;
  }
  fprintf(stderr, "I am process %ld, my parent is %ld\n",
                (long)getpid(), (long)getppid());
  return 0;
}
```

Σε διεργασίες χωρίς παιδιά επιστρέφει -1 και errno=ECHILD

Κλήσεις συστήματος
execl, execv, execlp, execvp

- `int execl(char *path, char *arg0, char *arg1, ... , char *argn, NULL)`
- `int execv(char *path, char *argv[])`
- `int execlp(char *path, char *arg0, char *arg1, ... , char *argn, NULL)`
- `int execvp(char *path, char *argv[])`
- Αντικαθιστούν μία διεργασία (κώδικα, δεδομένα, στοίβα) με αυτά του εκτελέσιμου αρχείου που δίνεται στο `path`
- Οι `execl` και `execv` απαιτούν απόλυτα ή σχετικά ονόματα-μονοπάτια στο `path`
- Οι `execlp` και `execvp` χρησιμοποιούν τη μεταβλητή περιβάλλοντος `PATH` για να βρουν το εκτελέσιμο αρχείο
- Οι `execl` και `execlp` θέλουν στο `arg0` το όνομα του εκτελέσιμου αρχείου, στα `arg1, ..., argn` τα ορίσματά του και το τελευταίο όρισμά τους `NULL`
- Οι `execv` και `execvp` θέλουν το όνομα του εκτελέσιμου αρχείου και τα ορίσματά του στα `argv[0], argv[1], ..., argv[n]` και `NULL` στο `argv[n+1]`
- Όλες οι κλήσεις της ομάδας `exec` επιστρέφουν λάθος (-1) αν δεν βρεθεί το εκτελέσιμο αρχείο, ενώ σε επιτυχία ποτέ δεν επιστρέφουν
- Απαίτηση: `#include <unistd.h>`

Παράδειγμα execl

```
/* File: execl_demo.c */
#include <stdio.h>                          /* For printf */
#include <unistd.h>                          /* For execl */
main()
{  int ret;
   printf("I am process %d and I will execute an 'ls -l ..'\n",
          getpid());
   ret = execl("/bin/ls", "ls", "-l", "..", NULL);
   if (ret == -1) {  /* Always true because of failure
                              to execute /bin/ls -l .. */
      perror("execl"); } }
```

```
$ ./execl_demo
I am process 1614 and I will execute an 'ls -l ..'
total 12
drwxr-xr-x    2 spro     users        4096 Jan 27 12:56 bin
drwxr-xr-x    2 spro     users        4096 Feb 15 22:24 c_progs
drwxr-xr-x    2 spro     users        4096 Feb 12 15:12 sh_scripts
$
```

Επειδή αντικαθίσταται ο κώδικας του προγράμματος με αυτόν της ls, εδώ φτάνει μόνο σε σφάλμα

Παράδειγμα execvp

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

```
$ ./execvp_demo
I am parent process 1615
I am child process 1616 and I will replace myself by 'date'
Fri Feb 15 22:26:26 EET 2002
Child terminated with exit code 0
$
```

Παράδειγμα

- *Να γραφεί ένα πρόγραμμα C που να δημιουργεί ένα δυαδικό δέντρο από διεργασίες το οποίο να έχει δεδομένο βάθος N. Κάθε διεργασία που δεν είναι φύλλο του δέντρου να εκτυπώνει την ταυτότητά της, την ταυτότητα του γονέα της καθώς και έναν αύξοντα αριθμό σύμφωνα με μία πρώτα κατά πλάτος διάσχιση του δέντρου.*

```
/* File: process_tree.c */
#include <stdio.h>                              /* For printf */
#include <stdlib.h>                               /* For exit */
main(int argc, char *argv[])
{  int i, depth, numb, pid1, pid2, status;
   if (argc > 1) depth = atoi(argv[1]);    /* Make integer */
   else          exit(0);
   if (depth > 5) {                      /* Avoid deep trees */
      printf("Depth should be up to 5\n"); exit(0); }
   numb = 1;            /* Holds the number of each process */
   for(i=1 ; i<=depth ; i++) {
      printf("I am process no %2d with PID %5d and PPID %5d\n",
             numb, getpid(), getppid());
      switch (pid1 = fork()) {
         case 0:                          /* Left child code */
            numb = 2*numb; break;
         case -1:              /* Error creating left child */
            perror("fork"); exit(1);
         default:                               /* Parent code */
            switch (pid2 = fork()) {
               case 0:                  /* Right child code */
                  numb = 2*numb+1; break;
               case -1:       /* Error creating right child */
                  perror("fork"); exit(1);
               default:                         /* Parent code */
                  wait(&status); wait(&status);
                  exit(0); } } } }
```

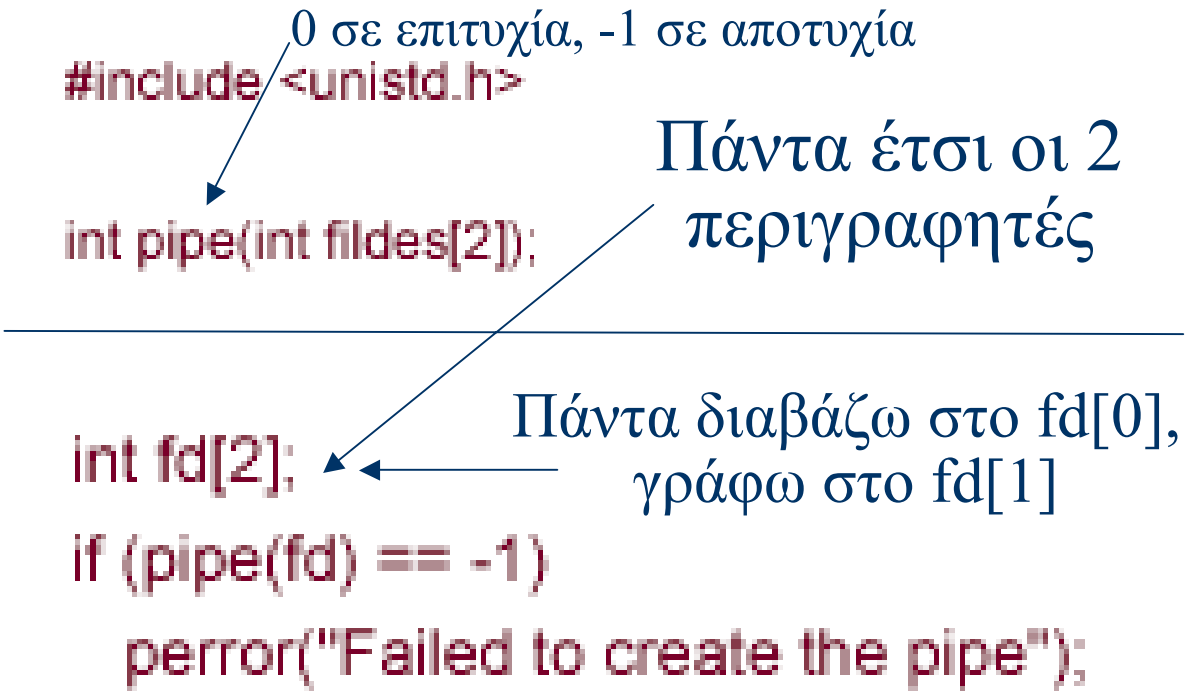
2 παιδιά ⟶

Εκτέλεση παραδείγματος

```
$ ./process_tree 1
I am process no  1 with PID  7608 and PPID  2133
$ ./process_tree 2
I am process no  1 with PID  7611 and PPID  2133
I am process no  2 with PID  7612 and PPID  7611
I am process no  3 with PID  7613 and PPID  7611
$ ./process_tree 3
I am process no  1 with PID  7618 and PPID  2133
I am process no  2 with PID  7619 and PPID  7618
I am process no  3 with PID  7620 and PPID  7618
I am process no  6 with PID  7623 and PPID  7620
I am process no  4 with PID  7621 and PPID  7619
I am process no  7 with PID  7624 and PPID  7620
I am process no  5 with PID  7622 and PPID  7619
$ ./process_tree 4
I am process no  1 with PID  7633 and PPID  2133
I am process no  2 with PID  7634 and PPID  7633
I am process no  4 with PID  7636 and PPID  7634
I am process no  8 with PID  7638 and PPID  7636
I am process no  3 with PID  7635 and PPID  7633
I am process no  7 with PID  7643 and PPID  7635
I am process no 14 with PID  7644 and PPID  7643
I am process no  6 with PID  7642 and PPID  7635
I am process no 12 with PID  7646 and PPID  7642
I am process no  9 with PID  7639 and PPID  7636
I am process no  5 with PID  7637 and PPID  7634
I am process no 10 with PID  7649 and PPID  7637
I am process no 11 with PID  7657 and PPID  7637
I am process no 13 with PID  7655 and PPID  7642
I am process no 15 with PID  7654 and PPID  7643
$
```

Σωλήνες (pipes)

- Χρήσιμα για επικοινωνία διεργασιών (συνήθως πατέρας-παιδί)
- Δύο άκρα: Ένα για γράψιμο, ένα για διάβασμα
  - Άρα και 2 περιγραφητές αρχείων



Παράδειγμα

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#define BUFSIZE 10

int main(void) {
   char bufin[BUFSIZE] = "empty";
   char bufout[] = "hello";
   int bytesin;
   pid_t childpid;
   int fd[2];

   if (pipe(fd) == -1) {
      perror("Failed to create the pipe");
      return 1;
   }
   bytesin = strlen(bufin);
   childpid = fork();
   if (childpid == -1) {
      perror("Failed to fork");
      return 1;
   }
   if (childpid)                          /* parent code */
      write(fd[1], bufout, strlen(bufout)+1);
   else                                   /* child code */
      bytesin = read(fd[0], bufin, BUFSIZE);
   fprintf(stderr, "[%ld]:my bufin is {%.*s}, my bufout is {%s}\n",
         (long)getpid(), bytesin, bufin, bufout);
   return 0;
}
```

Γράψιμο, διάβασμα

Συμπεριφορά read σε σωλήνες

- Αν όχι άδειος σωλήνας επιστρέφει αμέσως
- Αν άδειος σωλήνας, και κάποια διεργασία έχει ανοικτό το σωλήνα για εγγραφή, τότε μπλοκάρει
- Αν άδειος σωλήνας, και καμία διεργασία δεν έχει ανοικτό το σωλήνα για εγγραφή, τότε επιστρέφει 0

Παράδειγμα

```
/* File: pipe_demo.c */
#include <stdio.h>                          /* For printf */
#include <string.h>                         /* For strlen */
#include <stdlib.h>                           /* For exit */
#define READ  0                          /* Read end of pipe */
#define WRITE 1                         /* Write end of pipe */
char *phrase = "This is a test phrase.";
main()
{  int pid, fd[2], bytes;
   char message[100];
   if (pipe(fd) == -1) {                 /* Create a pipe */
      perror("pipe");
      exit(1); }
   if ((pid = fork()) == -1) {            /* Fork a child */
      perror("fork");
      exit(1); }
   if (pid == 0) {                        /* Child, writer */
      close(fd[READ]);                 /* Close unused end */
      write(fd[WRITE], phrase, strlen(phrase)+1);
      close(fd[WRITE]); }                /* Close used end */
   else {                                /* Parent, reader */
      close(fd[WRITE]);                /* Close unused end */
      bytes = read(fd[READ], message, sizeof(message));
      printf("Read %d bytes: %s\n", bytes, message);
      close(fd[READ]); } }               /* Close used end */
```

```
$ ./pipe_demo
Read 23 bytes: This is a test phrase.
$
```

Να κλείνετε τα άκρα που δεν χρησιμοποιείτε

Παράδειγμα 2

- *Να γραφεί ένα πρόγραμμα C που να συνδέει μέσω ενός σωλήνα την προκαθορισμένη έξοδο μίας εντολής με την προκαθορισμένη είσοδο μίας άλλης, να υλοποιηθεί δηλαδή μία σωλήνωση.*

```
/* File: connect.c */
#include <stdio.h>                          /* For printf */
#include <stdlib.h>                           /* For exit */
#include <unistd.h>                         /* For execlp */
#define READ  0                          /* Read end of pipe */
#define WRITE 1                         /* Write end of pipe */
main(int argc, char *argv[])
{  int fd[2], pid;
   if (pipe(fd) == -1) {                    /* Create a pipe */
      perror("pipe");
      exit(1); }
   if ((pid = fork()) == -1) {               /* Fork a child */
      perror("fork");
      exit(1); }
   if (pid != 0) {                          /* Parent, writer */
      close(fd[READ]);                     /* Close unused end */
      dup2(fd[WRITE], 1); /* Duplicate write end to stdout */
      close(fd[WRITE]);                     /* Close write end */
      execlp(argv[1], argv[1], NULL);    /* Execute argv[1] */
      perror("execlp"); }
   else {                                   /* Child, reader */
      close(fd[WRITE]);                    /* Close unused end */
      dup2(fd[READ], 0);    /* Duplicate read end to stdin */
      close(fd[READ]);                      /* Close read end */
      execlp(argv[2], argv[2], NULL);    /* Execute argv[2] */
      perror("execlp"); } }
```

Ότι γράφει το argv[1] πρόγραμμα γράφεται στο σωλήνα

Ότι διαβάζει το argv[2] πρόγραμμα διαβάζεται από το σωλήνα

Εκτέλεση παραδείγματος

```
$ ./connect who wc
      1       6      51
$ ./connect ls wc
     55      55     715
$ ./connect ls head
alarm_demo
alarm_demo.c
append_file
append_file.c
connect
connect.c
critical
critical.c
duplicate_fd
duplicate_fd.c
$ ./connect ps sort
$ 12947 pts/0    00:00:00 bash
13236 pts/0    00:00:00 ps
13237 pts/0    00:00:00 sort
  PID TTY          TIME CMD

$
```