

# Προγραμματισμός συστημάτων UNIX/POSIX

---

*Διαδικεργασιακή επικοινωνία: αγωγοί  
(IPC – inter-process communication: pipes)*



**ΜΥΥ 502**

- ❖ Κατά κάποιο τρόπο, θα δημιουργήσουμε ένα τύπο δυαδικού «αρχείου» στο οποίο θα έχουν πρόσβαση και οι δύο διεργασίες.
- ❖ Αρχικά, θα θεωρήσουμε ότι οι διεργασίες έχουν προέρθει από το ίδιο πρόγραμμα, με χρήση της `fork()` (μπορείτε να φανταστείτε γιατί;)
  - Διότι με την `fork` το παιδί κληρονομεί τα πάντα όπως είπαμε, ακόμα και τα αρχεία που έχει ανοίξει ο γονέας.
- ❖ Ο γονέας θα ανοίξει το αρχείο και το παιδί θα το έχει ήδη ανοιχτό όταν δημιουργηθεί με την `fork()`.
- ❖ Τα ειδικά αυτά αρχεία ονομάζονται **(ανώνυμοι) αγωγοί** (**unnamed pipes**)
  - Διότι κατά το άνοιγμα δεν δίνεται κανένα όνομα αρχείου.

# Αγωγοί (pipe)

## ❖ Δημιουργία & άνοιγμα pipe:

```
#include <unistd.h>  
int pipe(int pfd[2]);
```

## ❖ Επιστρέφει 0 αν όλα OK – αλλιώς αρνητικό.

## ❖ Στο pfd[] ανοίγουν και επιστρέφονται 2 περιγραφείς αρχείων.

- Με το **pfd[1]** μπορεί να γίνει **ΜΟΝΟ** γράψιμο (write) προς το pipe
- Με το **pfd[0]** θα μπορεί να γίνει **ΜΟΝΟ** διάβασμα (read) από το pipe
- Δηλαδή, ότι γράφεται στο pfd[1] διαβάζεται από το pfd[0].

## ❖ Εγγραφή και ανάγνωση γίνονται με τις γνωστές συναρτήσεις read() και write().

## ❖ Κλείσιμο γίνεται με την close().

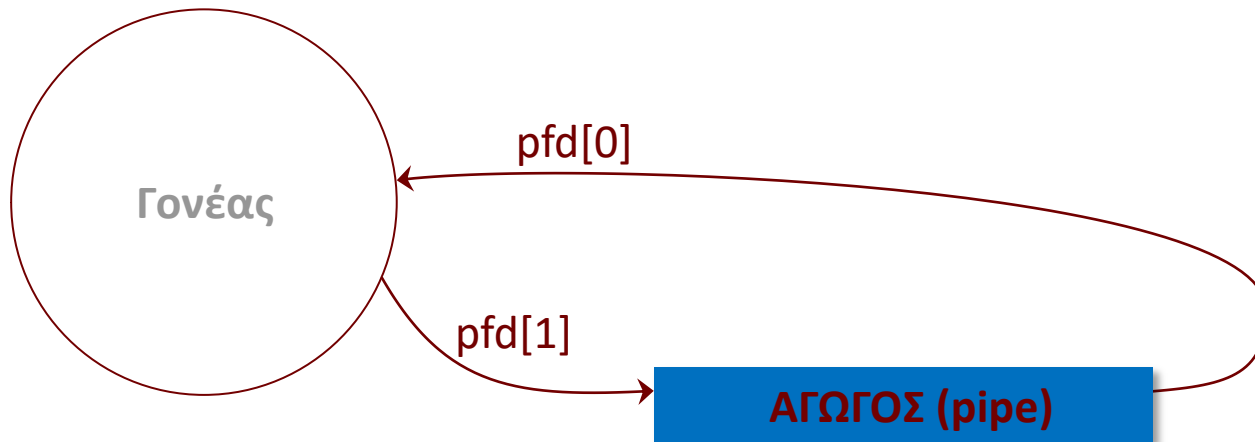
## ❖ Δεν μπορείτε να κάνετε lseek().

# Παράδειγμα με pipe

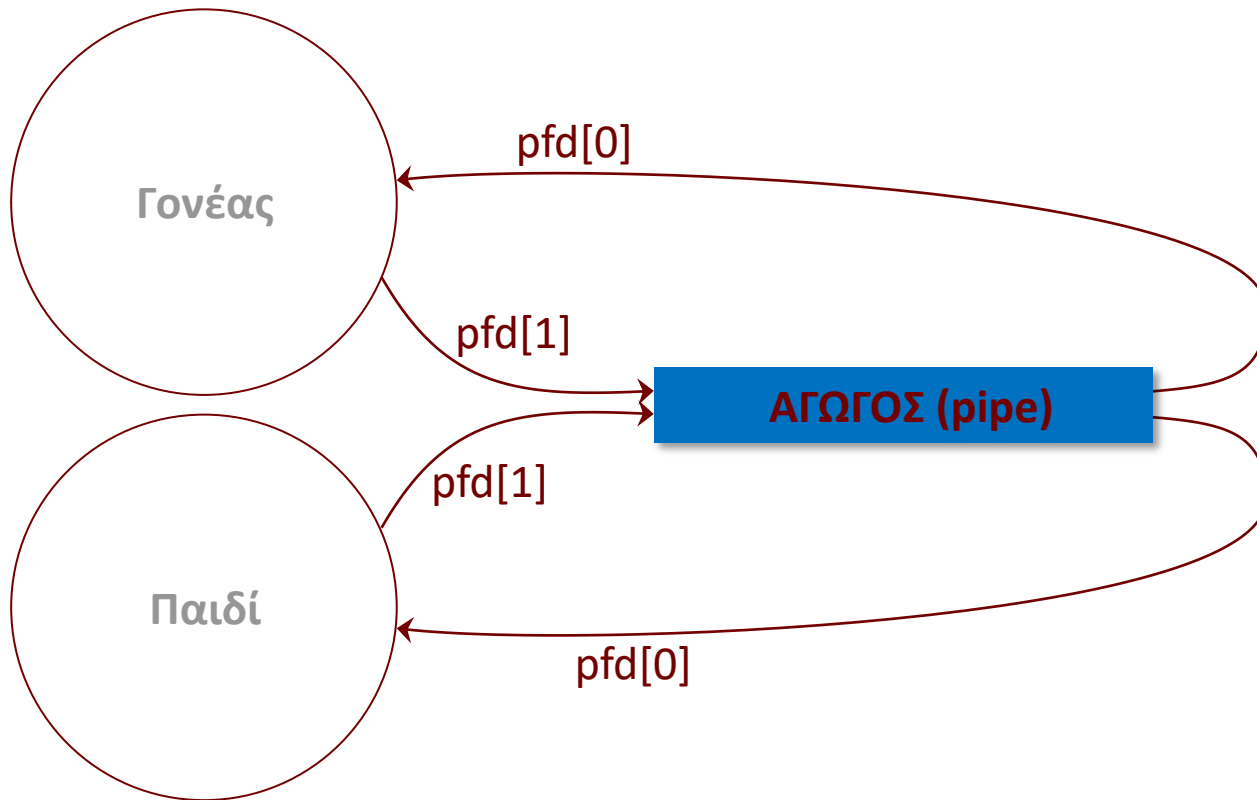
```
int main() {
    int  n, pfd[2];
    char msg[50];

    if (pipe(pfd) < 0) {
        perror("pipe()");
        exit(1);
    }
    if (fork() != 0) { /* Parent */
        close(pfd[0]); /* No reading by the parent */
        printf("(%d) - I am the parent; sending hello message.\n", getpid());
        write(pfd[1], "hello!!", 7);
    }
    else { /* Child */
        close(pfd[1]); /* No writing by the child */
        printf("(%d) - I am the child.\n", getpid());
        while ((n = read(pfd[0], msg, 2)) >= 0) {
            if (n == 0) break; /* EOF */
            printf("(%d) - read %d bytes from the pipe [%.s]\n", getpid(), n, n, msg);
        }
        if (n < 0) {
            perror("read() from child");
            exit(1);
        }
    }
    return 0; /* All files close */
}
```

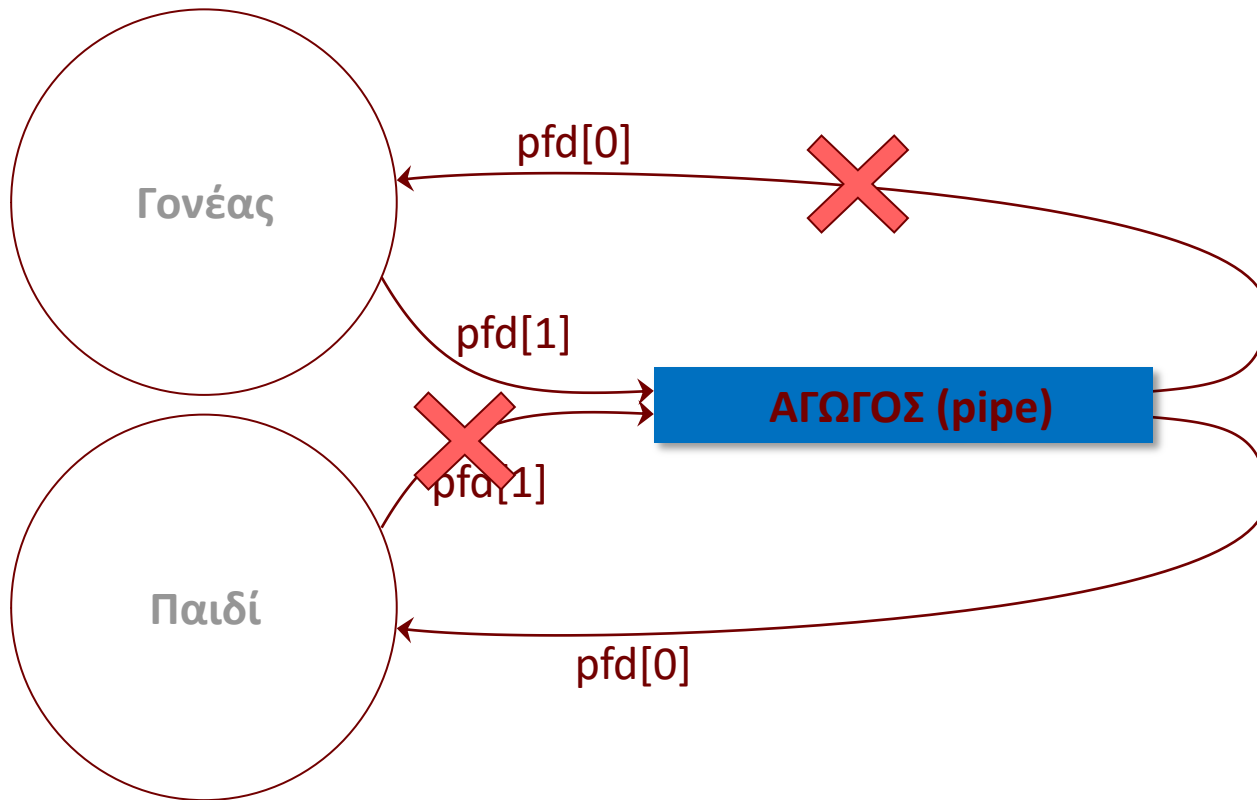
# Παράδειγμα – δημιουργία pipe



# Παράδειγμα – fork() και κληρονομιά pipe

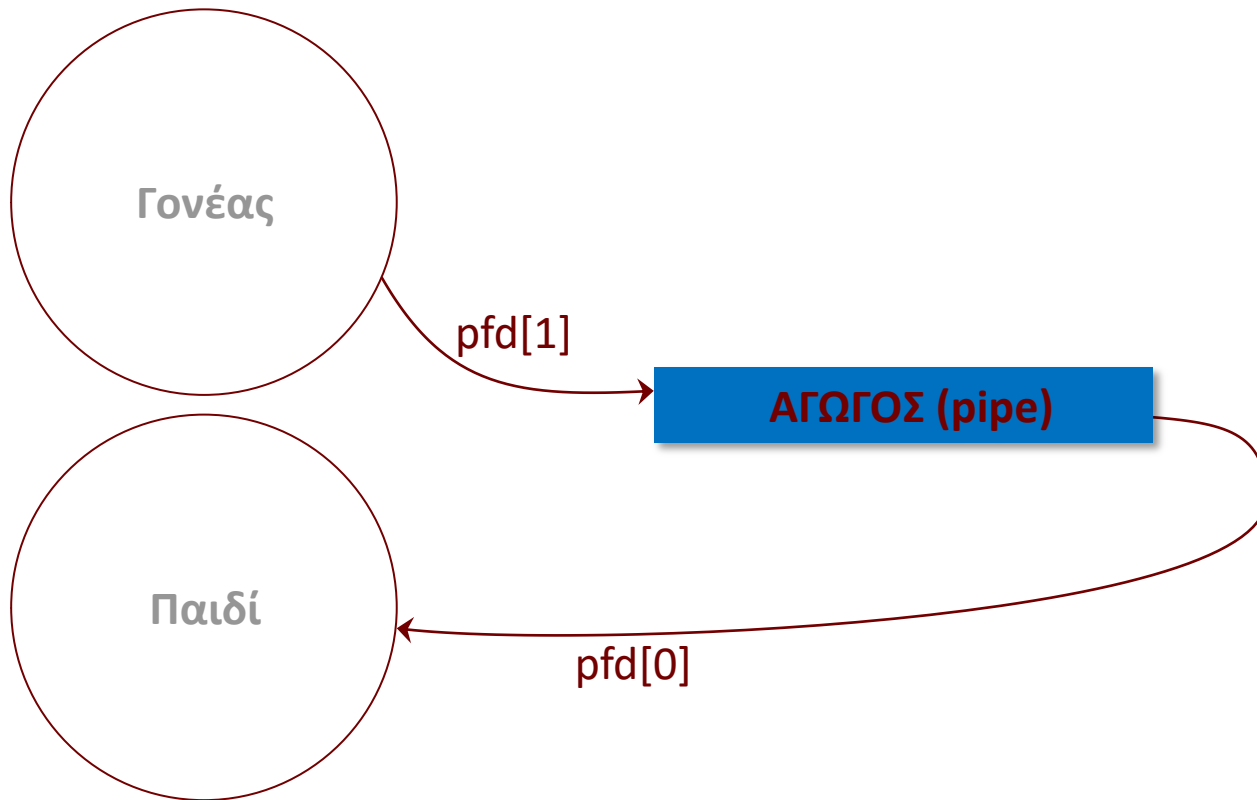


# Παράδειγμα – κλείσιμο άχρηστων περιγραφών



Πειράζει να μην κλείσω τους περιγραφείς;;;  
Δεν είναι απαραίτητο να κλείσουν! Όμως, ανάλογα  
με τι θέλουμε να κάνουνε, αν δεν κλείσουν μπορεί  
να έχουμε προβλήματα (βλ. παρακάτω).

# Παράδειγμα – τελική κατάσταση





## ❖ Εκτέλεση:

```
$ ./a.out
```

```
(23299) - I am the parent; sending hello message.
```

```
(23300) - I am the child.
```

```
(23300) - read 2 bytes from the pipe [he]
```

```
(23300) - read 2 bytes from the pipe [ll]
```

```
(23300) - read 2 bytes from the pipe [o!]
```

```
(23300) - read 1 bytes from the pipe [!]
```

## ❖ Μερικές λεπτομέρειες:

- **Όταν ο εγγραφέας κάνει close(),** στον αναγνώστη η read() θα επιστρέψει 0 (EOF-End of File), αφού όμως πρώτα διαβάσει όσα bytes είχαν ήδη γραφτεί.
  - ❖ Οποιοδήποτε δυαδικό αρχείο (άρα και ένας αγωγός) **παραμένει ανοικτό όσο υπάρχουν ανοικτοί περιγραφείς**
  - ❖ Για αυτό το παιδί κλείνει το rfd[1] όταν ξεκινάει – έτσι θα λάβει EOF μόλις κλείσει το rfd[1] και ο γονέας (εγγραφέας).
- Μπορεί να γράφει συνεχώς ο εγγραφέας ακόμα κι όταν δεν κάνει read() ο αναγνώστης;
  - ❖ Απάντηση: όχι – υπάρχει περιορισμένος χώρος (συνήθως 512 bytes). Αν γεμίσει, τότε ο εγγραφέας **μπλοκάρει** μέχρι ο αναγνώστης να αδειάσει χώρο με την read().

## ❖ Τι γίνεται αν το παιδί κάνει exec??

- Τι γίνεται με τα pipes που έχει ανοίξει ο πατέρας;
- Η διεργασία-παιδί τα κληρονομεί OK. Όμως τα γνωρίζει το νέο πρόγραμμα που θα εκτελεστεί με την exec και αν ναι, πώς;

## ❖ Απάντηση:

- Όπως είχαμε πει, οι κλήσεις τύπου exec δεν δημιουργούν νέα διεργασία, αλλά διατηρούν την υπάρχουσα η οποία απλά εκτελεί άλλο πρόγραμμα
- Πολλά πράγματα όπως το process id, το user id, ο τρέχων κατάλογος εργασίας κλπ διατηρούνται.
- Όπως επίσης και οι ανοιχτοί περιγραφείς αρχείων (file descriptors)
- Οι οποίοι περιλαμβάνουν και τους περιγραφείς των αγωγών!
- Επομένως, το πρόγραμμα που θα τρέξει μέσω της exec θα έχει ανοιχτούς τους ίδιους περιγραφείς αρχείων όπως η διεργασία που το εκτελεί, συμπεριλαμβανομένων και των αγωγών.

## ❖ Υπάρχει όμως ένα πρόβλημα:

- ΟΚ, ο αγωγός θα είναι ανοικτός, αλλά πώς θα γνωρίζει το νέο πρόγραμμα ποιοι ακριβώς είναι οι περιγραφείς για τον αγωγό; (ώστε να μπορεί να γράψει / διαβάσει από αυτούς;)

## ❖ Δηλαδή, στο προηγούμενο παράδειγμα:

```
if (fork() != 0) { /* Parent */
    close(pfd[0]); /* No reading by the parent */
    printf("(%d) - I am the parent; sending hello message.\n", getpid());
    write(pfd[1], "hello!!", 7);
}
else { /* Child */
    close(pfd[1]); /* No writing by the child */
    printf("(%d) - I am the child. I will exec program 'pipe2'.\n", getpid());
    execl("./otherprog", " otherprog", NULL);
    printf("execl() failed!\n", getpid());
}
}
```

πώς θα γνωρίζει το otherprog ποιο είναι το pfd[0] ???

# Περνώντας τον αγωγό μέσω exes

- ❖ Ο μόνος τρόπος να «περάσουμε» τον περιγραφέα στο πρόγραμμα που θα εκτελεστεί από την exes είναι:  
***μέσω ορισμάτων στην main του προγράμματος που θα εκτελεστεί***
- ❖ Το πρόγραμμα θα πρέπει να έχει ορίσματα στην main() του, τα οποία θα του δώσουν τον ακέραιο αριθμό που αντιπροσωπεύει τον περιγραφέα του αγωγού.
  - Μην ξεχνάμε ότι ο αγωγός είναι ήδη ανοιχτός λόγω της κληρονομιάς της διεργασίας-παιδιού, απλά το νέο πρόγραμμα δεν ξέρει τον αύξων αριθμό του (περιγραφέα).

## ❖ Γονική διεργασία & παιδί που εκτελεί exec:

```
int main() {
    int pfd[2];
    char s[10];

    if (pipe(pfd) < 0) {
        perror("pipe()");
        exit(1);
    }
    if (fork() != 0) { /* Parent */
        printf("(%d) - I am the parent; pfd[0]=%d,pfd[1]=%d.\n", getpid(),pfd[0],pfd[1]);
        close(pfd[0]); /* No reading from the parent */
        printf("(%d) - I am the parent; sending hello message.\n", getpid());
        write(pfd[1], "hello!!", 7);
    }
    else { /* Child */
        close(pfd[1]); /* No writing by the child */
        printf("(%d) - I am the child. I will exec otherprog.\n", getpid());
        sprintf(s, "%d", pfd[0]); /* Write pfd[0] to a string */
        execl("./ otherprog", otherprog, s, NULL); /* Pass s as an argument */
        printf("exec() failed!\n");
    }
    return 0;
}
```

# Παράδειγμα με exec & pipe, II

- ❖ Το otherprog.c (μετάφραση με: gcc -o otherprog otherprog.c):

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

int main(int argc, char *argv[]) {
    int pfd, n;
    char msg[10];

    if (argc != 2)
        exit(1);
    pfd = atoi(argv[1]); /* Get the file descriptor */
    printf("(%d) - I am %s. I will read from fd %d\n", getpid(), argv[0], pfd);

    while ((n = read(pfd, msg, 2)) >= 0) {
        if (n == 0) break; /* EOF */
        printf("(%s) - read %d bytes from the pipe [%.s]\n", argv[0], n, n, msg);
    }
    if (n < 0) {
        perror("read() from child");
        exit(1);
    }
    return 0;
}
```

# Εκτέλεση παραδείγματος

```
$ ./a.out
(23859) - I am the parent; pfd[0]=3, pdf[1]=4.
(23859) - I am the parent; sending hello message.
(23860) - I am the child. I will exec otherprog.
(23860) - I am otherprog. I will read from fd 3
(pipe2) - read 2 bytes from the pipe [he]
(pipe2) - read 2 bytes from the pipe [ll]
(pipe2) - read 2 bytes from the pipe [o!]
(pipe2) - read 1 bytes from the pipe [!]
```

1. Ο γονέας δημιουργεί και ανοίγει έναν αγωγό με την `pipe()`.
2. Επιστρέφονται 2 περιγραφείς, στον έναν γίνεται εγγραφή και από τον άλλον διάβασμα.
3. Δημιουργείται διεργασία-παιδί με τη `fork()`. Το παιδί κληρονομεί τους περιγραφείς.
4. Άμεσο κλείσιμο όσων περιγραφέων δεν χρειαζόμαστε (ώστε να ληφθούν EOF όταν πρέπει).
5. Διάβασμα και γράψιμο γίνεται σαν να πρόκειται για απλό αρχείο, μέσω `read()` και `write()`.
6. Κάθε διεργασία κλείνει με `close()`.
7. Αν το παιδί εκτελέσει μέσω `exec` κάποιο άλλο πρόγραμμα, οι περιγραφείς μπορούν να περάσουν στο δεύτερο πρόγραμμα μέσω ορισμάτων στη `main()` του.



# Θέματα με τους αγωγούς: αμφίδρομη επικοινωνία?

- ❖ Εκτός από τον γονέα, μπορεί να γράφει και το παιδί στον αγωγό για να τα διαβάσει ο γονέας;
- ❖ *Απάντηση:*  
η σύντομη και ασφαλής απάντηση είναι: καλύτερα **ΌΧΙ**,
  - παρότι υπό προϋποθέσεις μπορεί και να γίνεται.
- ❖ Οι αγωγοί θεωρούνται **μονό-κατευθυντήριες** επικοινωνίες.
- ❖ Πώς μπορεί λοιπόν να γράφει ο πατέρας προς το παιδί αλλά και το παιδί να γράφει προς τον πατέρα;
- ❖ *Απάντηση:*  
Πρέπει να δημιουργηθούν **δύο ανεξάρτητοι αγωγοί**:
  - στον έναν θα γράφει ο πατέρας και θα διαβάζει το παιδί και
  - στον άλλον θα γράφει το παιδί και θα διαβάζει ο γονέας.

# Θέματα με τους αγωγούς: αν δεν ελέγχουμε το *exes*?

❖ Όπως είπαμε πριν:

7. Αν το παιδί εκτελέσει μέσω *exes* κάποιο άλλο πρόγραμμα, οι περιγραφείς μπορούν να περάσουν στο δεύτερο πρόγραμμα μέσω ορισμάτων στη *main()* του.

Τι γίνεται αν το πρόγραμμα που εκτελείται μέσω της *exes* είναι μία έτοιμη εφαρμογή που δεν έχουμε πρόσβαση στον κώδικά της; Πώς μπορεί να ξέρει τον αγωγό μας ώστε να επικοινωνήσουμε μαζί της;

❖ Απάντηση:

Βασικά, δεν μπορούμε να κάνουμε τίποτε!

- ❖ Ξέρουμε κάτι για κάθε εφαρμογή, ακόμα και για αυτές που δεν έχουμε γράψει εμείς;
- ❖ Μήπως **όλες** οι εφαρμογές γράφουν κάπου και διαβάζουν από κάπου;
- ❖ *Απάντηση:*  
Πάντα έχουν ανοιχτά (υποχρεωτικά) δύο συγκεκριμένα αρχεία,
  - το standard input (με file descriptor 0, για διάβασμα από το πληκτρολόγιο) και
  - το standard output (με file descriptor 1, για γράψιμο στην οθόνη).
- ❖ Για λόγους φορητότητας (portability), αντί να χρησιμοποιούμε το 0 για το standard input και 1 για το standard output, πρέπει να χρησιμοποιούμε τα **STDIN\_FILENO** και **STDOUT\_FILENO** αντίστοιχα.

- ❖ Από μόνο του δεν βοηθάει. Όμως υπάρχει η εξής ιδέα:
  - μπορούμε με κάποιον τρόπο να «συνδέσουμε» τους αγωγούς μας με την standard input ή/και την standard output της νέας διεργασίας;
  - Έτσι η διεργασία θα πιστεύει ότι διαβάζει π.χ. από το πληκτρολόγιο αλλά στην πράξη τα δεδομένα θα της τα δίνει ο γονέας μέσω ενός αγωγού.
- ❖ *Απάντηση:*

Αυτό όντως θα λειτουργήσει. Αρκεί βέβαια να μπορούμε να το κάνουμε!
- ❖ Απαιτείται να μπορούμε να κάνουμε «περίεργες» αλλαγές στους περιγραφείς...

# Προγραμματισμός συστημάτων UNIX/POSIX

---

*«Παιχνίδια» με περιγραφείς αρχείων &  
αγωγούς*



# «Παιχνίδια» με τους περιγραφείς αρχείων

- ❖ Υπάρχει η δυνατότητα να δημιουργήσουμε έναν **νέο (αντίγραφο) περιγραφέα** ο οποίος μπορεί να χρησιμοποιηθεί για προσπέλαση ενός **ήδη ανοιχτού αρχείου**.
- ❖ Δηλαδή, το ίδιο αρχείο μπορούμε να το προσπελάσουμε από δύο διαφορετικούς / ανεξάρτητους περιγραφείς.
- ❖ Μάλιστα μπορούμε να κλείσουμε τον πρώτο αλλά το αρχείο θα παραμείνει ανοιχτό μιας και είναι ανοιχτός ο δεύτερος.
- ❖ Δύο είναι οι κλήσεις που πετυχαίνουν τα παραπάνω:
  - `dup( )`
  - `dup2( )`

# Η κλήση dup()

- ❖ Δημιουργία αντιγράφου περιγραφέα (υποτίθεται ότι το αρχείο με περιγραφέα fd είναι ήδη ανοιχτό):

```
#include <unistd.h>
int dup(int fd);
```

- ❖ Επιστρέφει έναν νέο περιγραφέα, ο οποίος μπορεί να χρησιμοποιηθεί να προσπελάσουμε το ίδιο αρχείο με τον fd.
- ❖ Για παράδειγμα, το παρακάτω γράφει στην οθόνη:

```
if ((new = dup(STDOUT_FILENO)) > 0) {
    write(new, "Hi!\n", 4);
    close(new);
}
```

# Η κλήση dup2()

- ❖ Δημιουργία συγκεκριμένου αντιγράφου περιγραφέα (υποτίθεται ότι το αρχείο με περιγραφέα fd είναι ήδη ανοιχτό):

```
#include <unistd.h>
int dup2(int fd, int newfd);
```

- ❖ Ο νέος περιγραφέας θα είναι αυτός που δίνουμε (newfd)
- ❖ Επιστρέφει τον νέο περιγραφέα (δηλαδή το newfd) αν όλα πάνε καλά, αλλιώς έναν αρνητικό αριθμό.
- ❖ **Αν ο newfd χρησιμοποιείται ήδη, η dup2() πρώτα κλείνει το αρχείο του και στη συνέχεια κάνει τον newfd να προσπελαύνει το αρχείο του fd.**
- ❖ Για παράδειγμα, το παρακάτω γράφει στην οθόνη μέσω του 8:

```
if (dup2(STDOUT_FILENO, 8) > 0) {
    write(8, "Hi!\n", 4);
    close(8);
}
```



# Παράδειγμα dup / dup2

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
```

```
int main() {
    int out, bak;
```

```
    bak = dup(STDOUT_FILENO);    /* Backup standard output */
    out = open("file.txt", O_CREAT | O_TRUNC | O_RDWR, S_IRUSR | S_IWUSR);
    if (bak < 0 || out < 0) exit(1);
```

```
    printf("Hello world 1 \n");
```

```
    dup2(out, STDOUT_FILENO);    /* out will be the new standard output */
    close(out);                  /* No longer needed */
```

```
    printf("Hello world 2 \n");
```

```
    dup2(bak, STDOUT_FILENO);    /* close & restore the original standard output */
    close(bak);                  /* No longer needed */
```

```
    printf("Hello world 3 \n");
    return 0;
```

```
}
```

## Εκτέλεση:

```
$ ./a.out
Hello world 1
Hello world 3
$ cat file.txt
Hello world 2
```

## ❖ Γονική διεργασία & παιδί που εκτελεί exec:

```
int main() {
    int pfd[2];

    if (pipe(pfd) < 0) {
        perror("pipe()");
        exit(1);
    }
    if (fork() != 0) { /* Parent */
        printf("(%d) - I am the parent; pfd[0]=%d,pdf[1]=%d.\n", getpid(),pfd[0],pfd[1]);
        close(pfd[0]); /* No reading from the parent */
        printf("(%d) - I am the parent; sending hello message.\n", getpid());
        write(pfd[1], "hello!!", 7);
    }
    else { /* Child */
        close(pfd[1]); /* No writing by the child */
        if (dup2(pfd[0], STDIN_FILENO) >= 0) { /* Duplicate pipe to 0 (std input!) */
            execl("./pipe4", "pipe4", NULL);
            printf("exec() failed!\n");
        }
        perror("dup2");
    }
    return 0;
}
```

❖ Το pipe4.c (μετάφραση με: `gcc -o pipe4 pipe4.c`):

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

int main() {
    int  n;
    char msg[10];

    printf("(%d) - I am pipe4. I will read standard input.\n", getpid());

    while ((n = read(STDIN_FILENO, msg, 2)) >= 0) {
        if (n == 0) break; /* EOF */
        printf("(pipe4) - read %d bytes [%.*s]\n", n, n, msg);
    }
    if (n < 0) {
        perror("read() from child");
        exit(1);
    }
    return 0;
}
```

# Εκτέλεση παραδείγματος

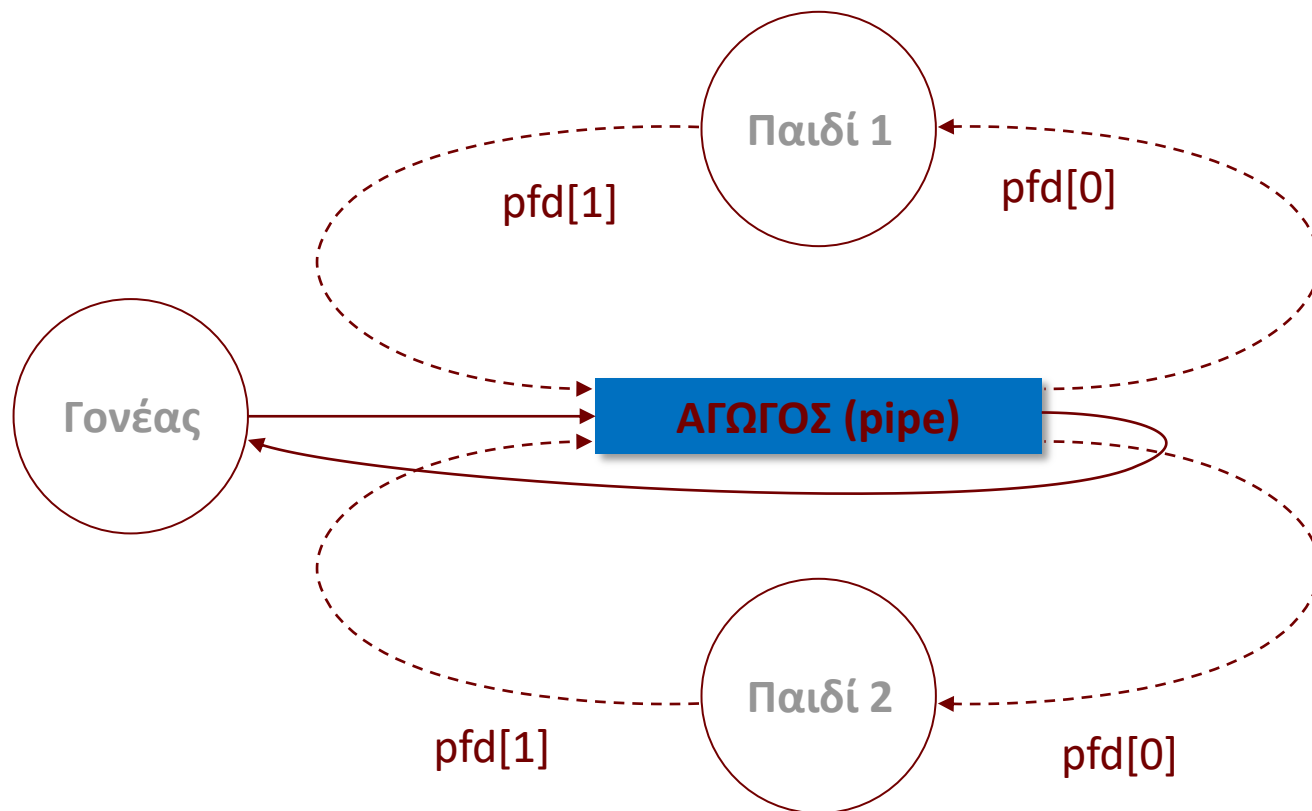
```
$ ./a.out
(24365) - I am the parent; pfd[0]=3, pdf[1]=4.
(24365) - I am the parent; sending hello message.
(24366) - I am the child. I will exec pipe4.
(24366) - I am pipe4. I will read standard input.
(pipe4) - read 2 bytes [he]
(pipe4) - read 2 bytes [ll]
(pipe4) - read 2 bytes [o!]
(pipe4) - read 1 bytes [!]
```



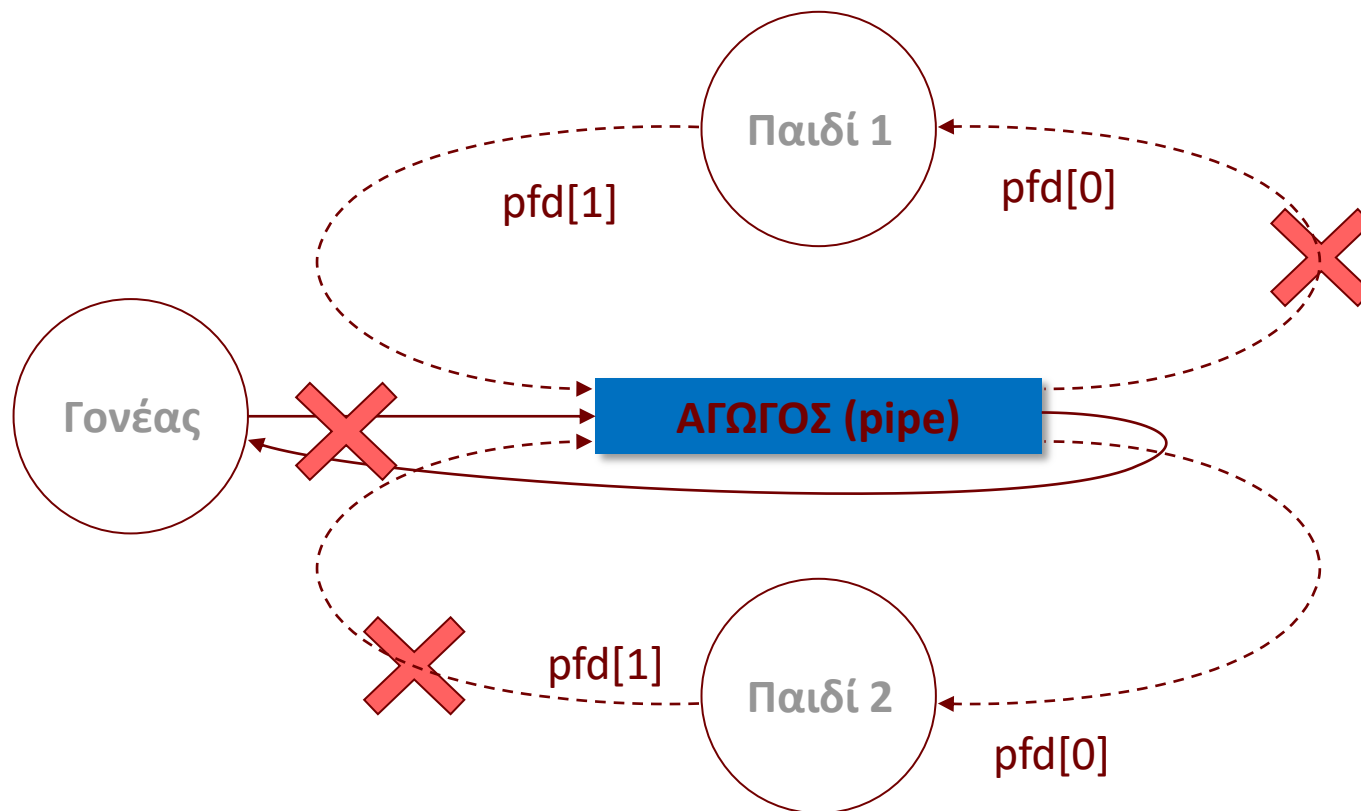
# Επιπλέον παράδειγμα: 2 παιδιά

- ❖ Δημιουργήστε πρόγραμμα όπου η γονική διεργασία δημιουργεί **δύο (2)** παιδιά, όπου το πρώτο εκτελεί την εντολή “ls -l” και, μέσω αγωγού, στέλνει την έξοδο της εντολής στο δεύτερο παιδί, το οποίο εκτελεί την εντολή “wc”.
- ❖ Λύση:
  - Ο πατέρας θα δημιουργήσει έναν αγωγό
  - Στη συνέχεια θα δημιουργήσει 2 παιδιά
  - Τα παιδιά θα αντιγράψουν τα κατάλληλα άκρα του αγωγού με dup2() στο stdin/stdout
  - Τα παιδιά θα κάνουν exec.

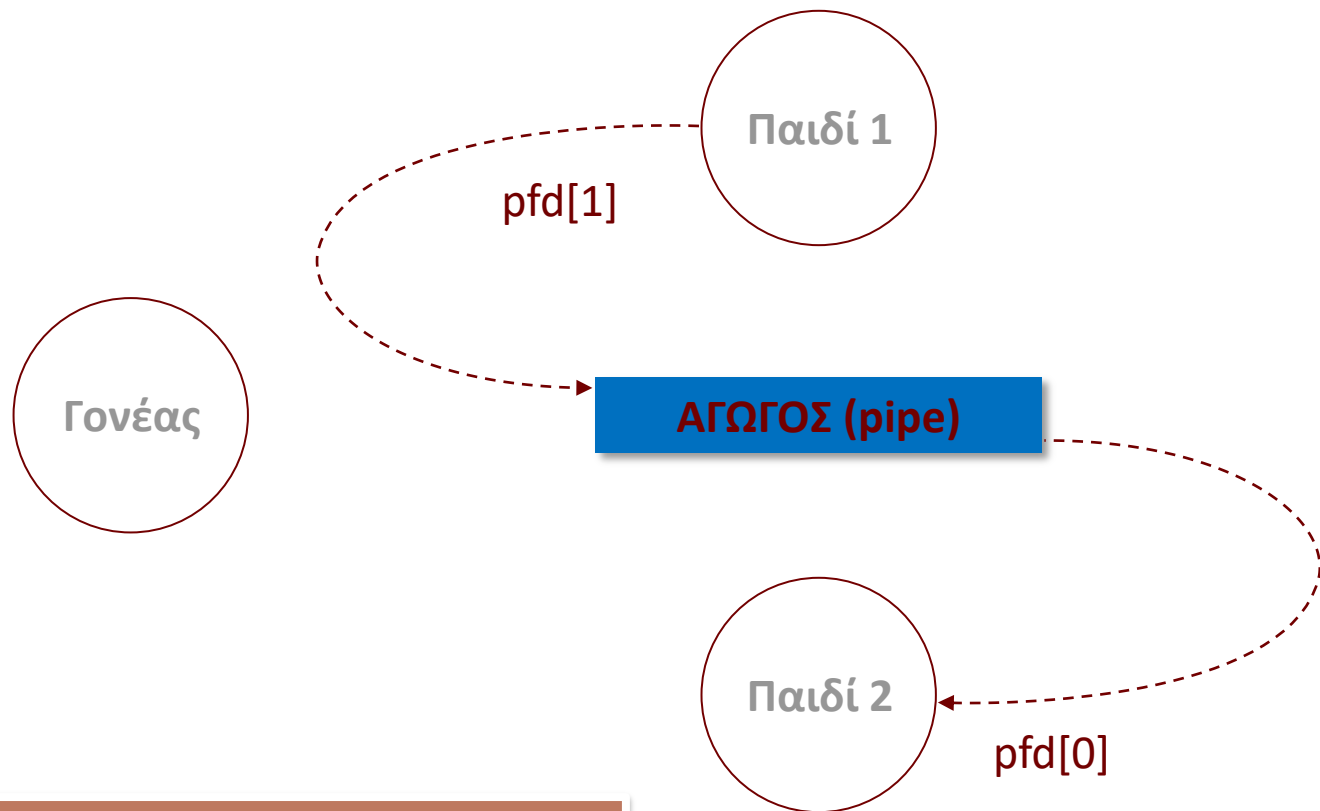
# Σχηματισμός αγωγού



# Σχηματισμός αγωγού



# Σχηματισμός αγωγού



## Υπενθύμιση:

θα πείραζε αν ΔΕΝ κλείναμε τους περιγραφείς  
και τους αφήναμε όλους ανοιχτούς;



# Ο κώδικας

```
int main() {
    int pfd[2];

    if (pipe(pfd) < 0) {
        perror("pipe()"); exit(1);
    }

    if (fork() == 0) {        /* child 1 */
        close(pfd[0]);        /* No reading from the pipe */
        if (dup2(pfd[1], STDOUT_FILENO) >= 0) {    /* Dup to std output */
            close(pfd[1]);    /* No longer needed! */
            execl("/bin/ls", "ls", "-l", NULL);
            printf("exec() failed from child 1!\n");
        }
        perror("dup2 from child 1"); exit (1);
    }

    if (fork() == 0) {        /* child 2 */
        close(pfd[1]);        /* No writing to the pipe */
        if (dup2(pfd[0], STDIN_FILENO) >= 0) {    /* Dup to std input */
            close(pfd[0]);    /* No longer needed! */
            execl("/usr/bin/wc", "wc", NULL);
            printf("exec() failed from child 2!\n");
        }
        perror("dup2 from child 2"); exit (1);
    }

    ...                      /* Parent code */
    return 0;
}
```

Είναι κάτι που πρέπει να κάνει ο γονέας αν θέλει να περιμένει να τελειώσουν τα παιδιά του;