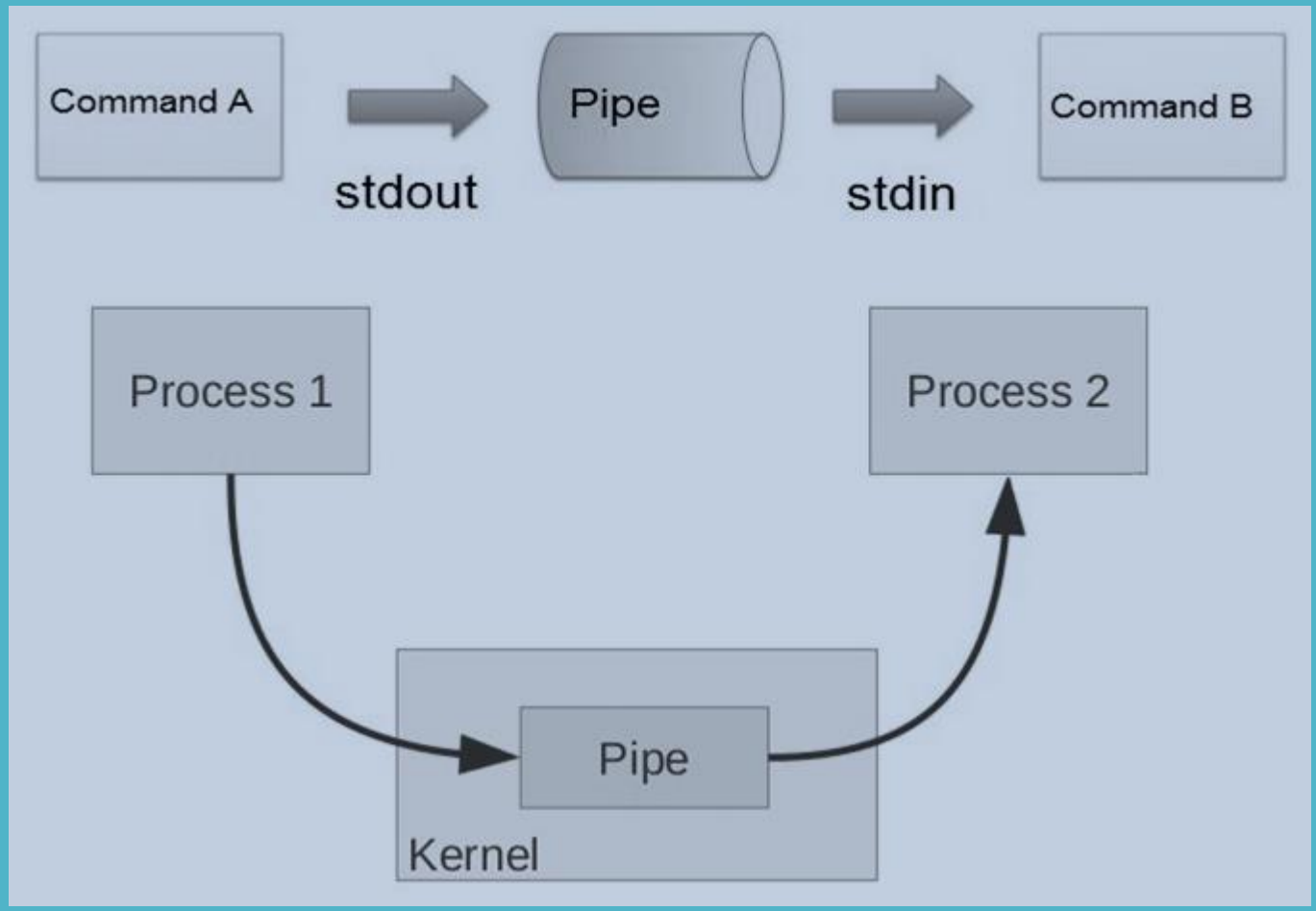
Αγωγοί


Command A
stdout
Pipe
stdin
Command B
Process 1
Process 2
Pipe
Kernel

Αγωγοί

Οι αγωγοί (pipes) αποτελούν τον έναν από τους αρκετούς τρόπους επικοινωνίας μεταξύ διεργασιών (οι άλλες λύσεις είναι η χρήση κοινόχρηστης μνήμης, οι ουρές μηνυμάτων και οι υποδοχείς).

Αποτελούν κανάλι επικοινωνίας που συνδέει την έξοδο της μιας διεργασίας με την είσοδο της άλλης.

Η επικοινωνία είναι απλής κατεύθυνσης (unnamed pipes) ή (σπάνια) διπλής κατεύθυνσης (named pipes) – υποστηρίζεται μόνο από λίγα συστήματα, σχεδόν όλα προσφέρουν απλής κατεύθυνσης.

Στο ένα άκρο υπάρχει η διεργασία συγγραφέας που στέλνει την έξοδό της στον αγωγό.

Στο άλλο άκρο υπάρχει η διεργασία αναγνώστης που διαβάζει την είσοδό της από τον αγωγό.

Οι αγωγοί διαθέτουν το δικό τους σύστημα αρχείων (pipefs) που δεν προσαρτάται κάτω από τον ριζικό κατάλογο αλλά παράπλευρα με αυτό.

Δεν είναι δυνατή η άμεση εξέτασή τους από το χρήστη όπως συμβαίνει με τα συστήματα αρχείων.

Το σύστημα pipefs αποθηκεύεται ως ένα IMFS (In-Memory File System) το οποίο ως δανείζεται τους πόρους του για την αποθήκευση αρχείων και καταλόγων όχι από το δίσκο αλλά από τη μνήμη.

Η χωρητικότητα ενός αγωγού είναι περιορισμένη (65536 bytes από τον πυρήνα 2.6.11 και μετά).

Αγωγοί

Απαιτείται συγχρονισμός διεργασιών → αν ο αναγνώστης προσπεράσει το συγγραφέα, αναστέλλει τη λειτουργία του μέχρι να υπάρξουν περισσότερα δεδομένα, ενώ το ίδιο ισχύει και όταν ο συγγραφέας προσπεράσει τον αναγνώστη.

Ο αναγνώστης αναστέλλει τη λειτουργία του όταν ο αγωγός είναι άδειος, αναμένοντας νέα δεδομένα.

Ο συγγραφέας αναστέλλει τη λειτουργία του όταν ο αγωγός είναι πλήρης, αναμένοντας την ανάγνωση δεδομένων από τον αναγνώστη.

Όταν κλείσουν όλοι οι περιγραφείς αρχείων που σχετίζονται με το άκρο του συγγραφέα, η ανάγνωση δεδομένων από τον αγωγό επιστρέφει τον κωδικό EOF.

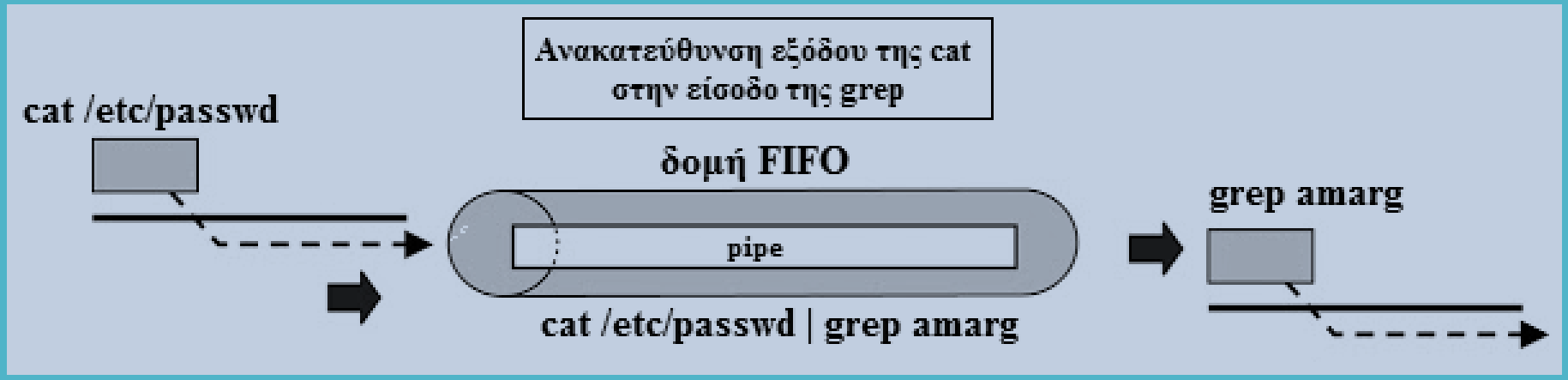
Η βέλτιστη απόδοση επιτυγχάνεται όταν ο αναγνώστης διαβάζει δεδομένα με την ίδια ταχύτητα με την οποία ο συγγραφέας γράφει δεδομένα.

Οι αγωγοί υλοποιούνται τόσο σε επίπεδο φλοιού όσο και σε επίπεδο πυρήνα αν και οι λειτουργίες που προσφέρει ο πυρήνας είναι πολύ πιο γενικές σε σχέση με αυτές που προσφέρει ο φλοιός.

Οι αγωγοί δεν είναι κανονικά αρχεία (δεν εμφανίζονται στο σύστημα αρχείων αν και διαθέτουν i-node) και δεν υπάρχει σύνδεσμος προς αυτούς (είναι εσωτερικά αντικείμενα του πυρήνα).

Αγωγοί

Παράδειγμα χρήσης αγωγού ανάμεσα στις εντολές cat και grep



Η ανακατεύθυνση της εξόδου της cat στην είσοδο της grep γίνεται με δύο τρόπους:

Χρησιμοποιώντας ενδιάμεσο αρχείο στο οποίο η cat αποθηκεύει την έξοδό της και από το οποίο η grep διαβάζει την είσοδό της, δηλαδή ως

```
cat /etc/passwd > tempFile
grep amarg < tempFile
```

ή χρησιμοποιώντας αγωγό ως

```
cat /etc/passwd | grep amarg < tempFile
```

Στη δεύτερη περίπτωση δεν απαιτείται η χρήση του ενδιάμεσου αρχείου tempFile.

Αγωγοί

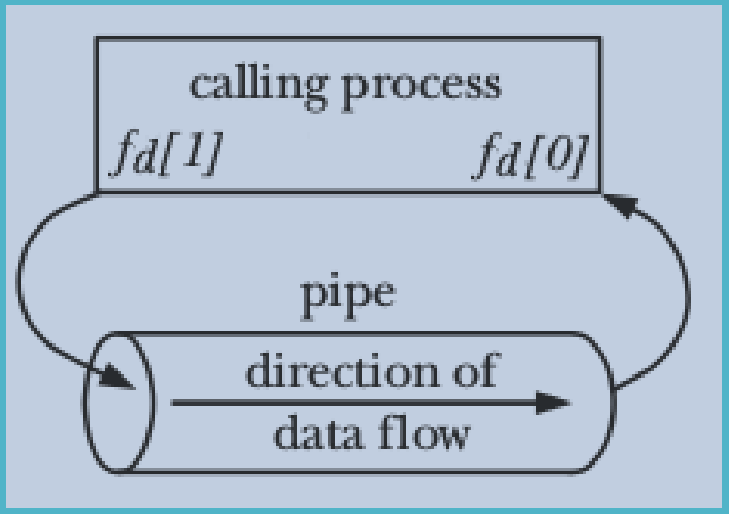
Η δημιουργία ενός αγωγού στηρίζεται στη χρήση της συνάρτησης pipe (unistd.h)

int pipe (int fd [2]);

Αυτή η συνάρτηση δημιουργεί έναν αγωγό και επιστρέφει στο όρισμα fd (που είναι πίνακας δύο ακεραίων) τους δύο περιγραφείς αρχείων που σχετίζονται με τα δύο άκρα του αγωγού.

Ο περιγραφέας αρχείου fd [0] σχετίζεται με το άκρο του αναγνώστη (read).

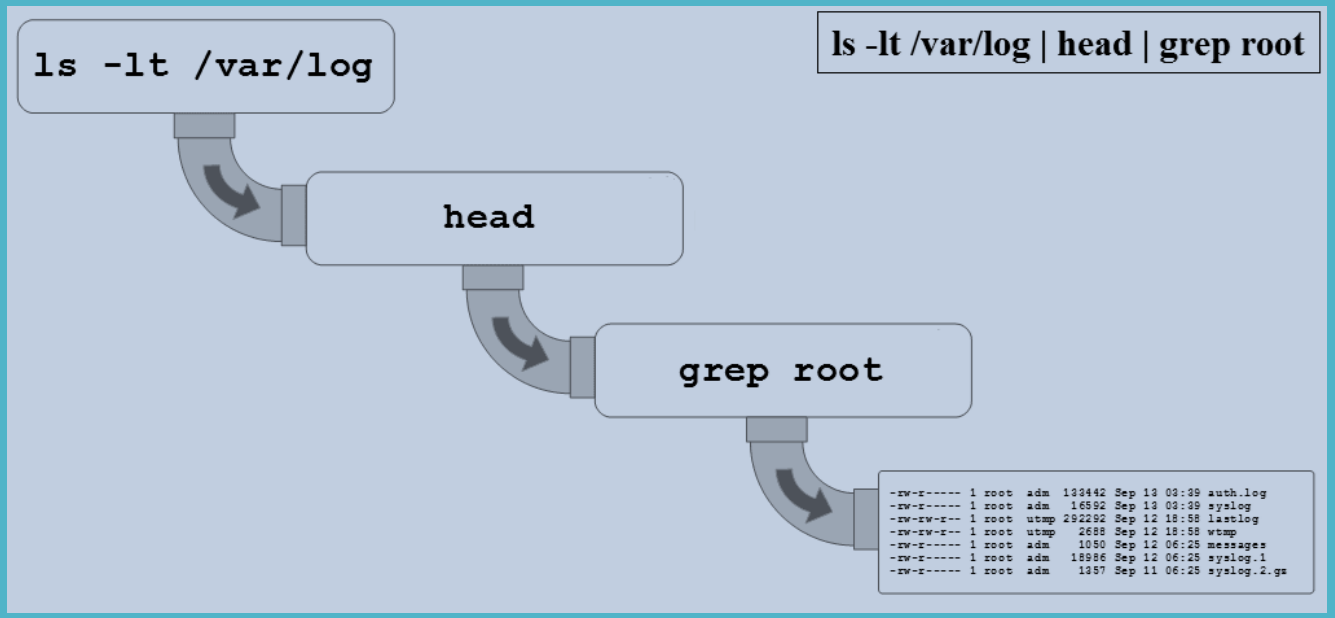
Ο περιγραφέας αρχείου fd [1] σχετίζεται με το άκρο του συγγραφέα (write).



Οι περιγραφείς αρχείου που δεν χρησιμοποιούνται μπορούν να κλείσουν με τη συνάρτηση close.

Αγωγοί

ΠΛΕΟΝΕΚΤΗΜΑ → οι διεργασίες εκτελούνται ταυτόχρονα και όχι ακολουθιακά η μία μετά την άλλη όπως στην περίπτωση χρήσης προσωρινού αρχείου. Όλα τα δεδομένα διακινούνται μέσω του πυρήνα.

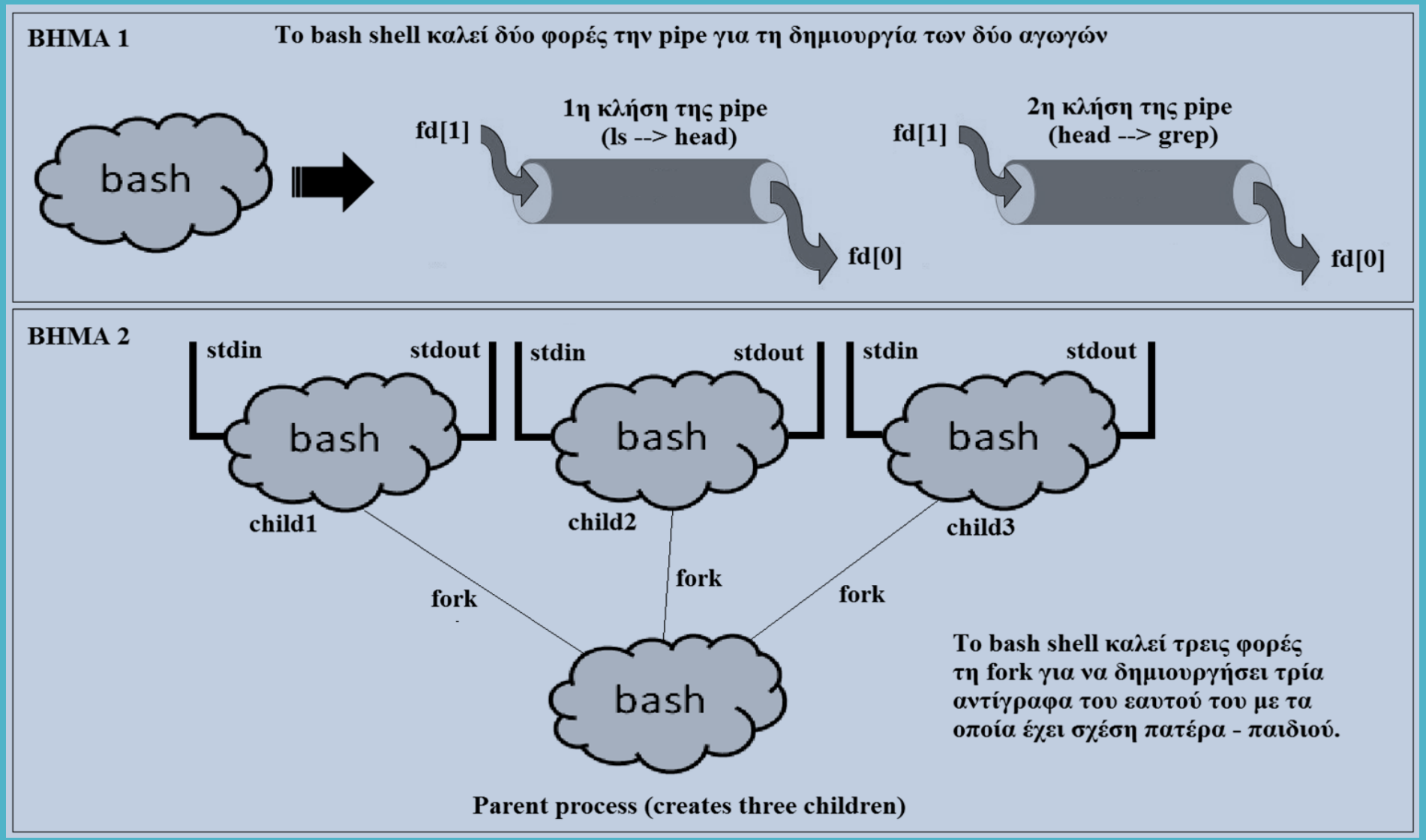


ΜΕΙΟΝΕΚΤΗΜΑ → οι διεργασίες πρέπει να σχετίζονται με σχέση γονέα – παιδιού ή να αποτελούν αμφότερες παιδιά του ιδίου γονέα. Αυτό είναι περιοριστικό για εφαρμογές client / server.

ΜΕΙΟΝΕΚΤΗΜΑ → Σε περιπτώσεις εγγραφών πολύ μεγάλου μεγέθους, δεν διασφαλίζεται η ατομικότητα και εάν υπάρχουν πολλοί συγγραφείς ενδέχεται να επέλθει σύγχυση.

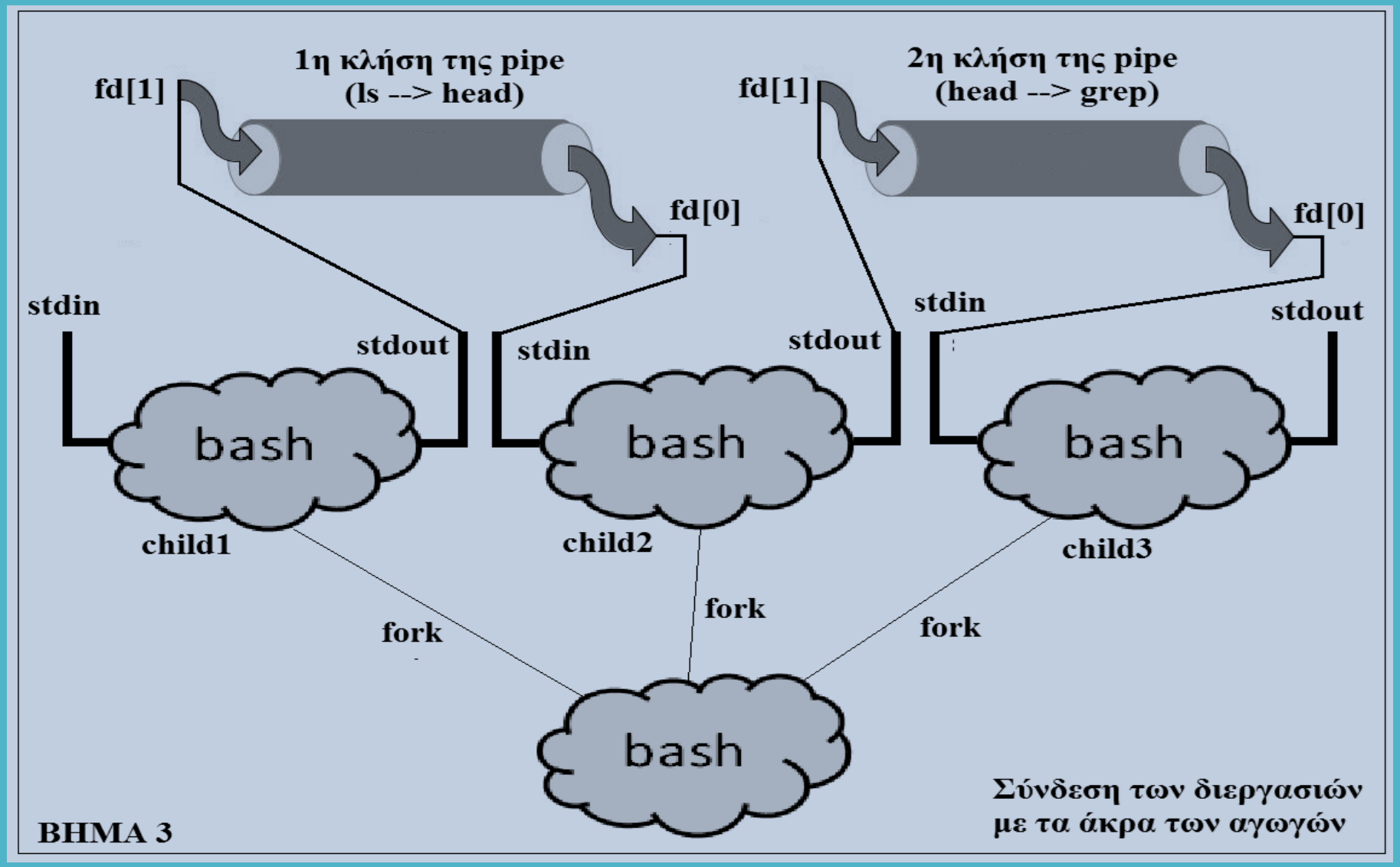
Αγωγοί

Η διαδικασία δημιουργίας και χρήσης αγωγών από το φλοιό, αποτελείται από τέσσερα βήματα:



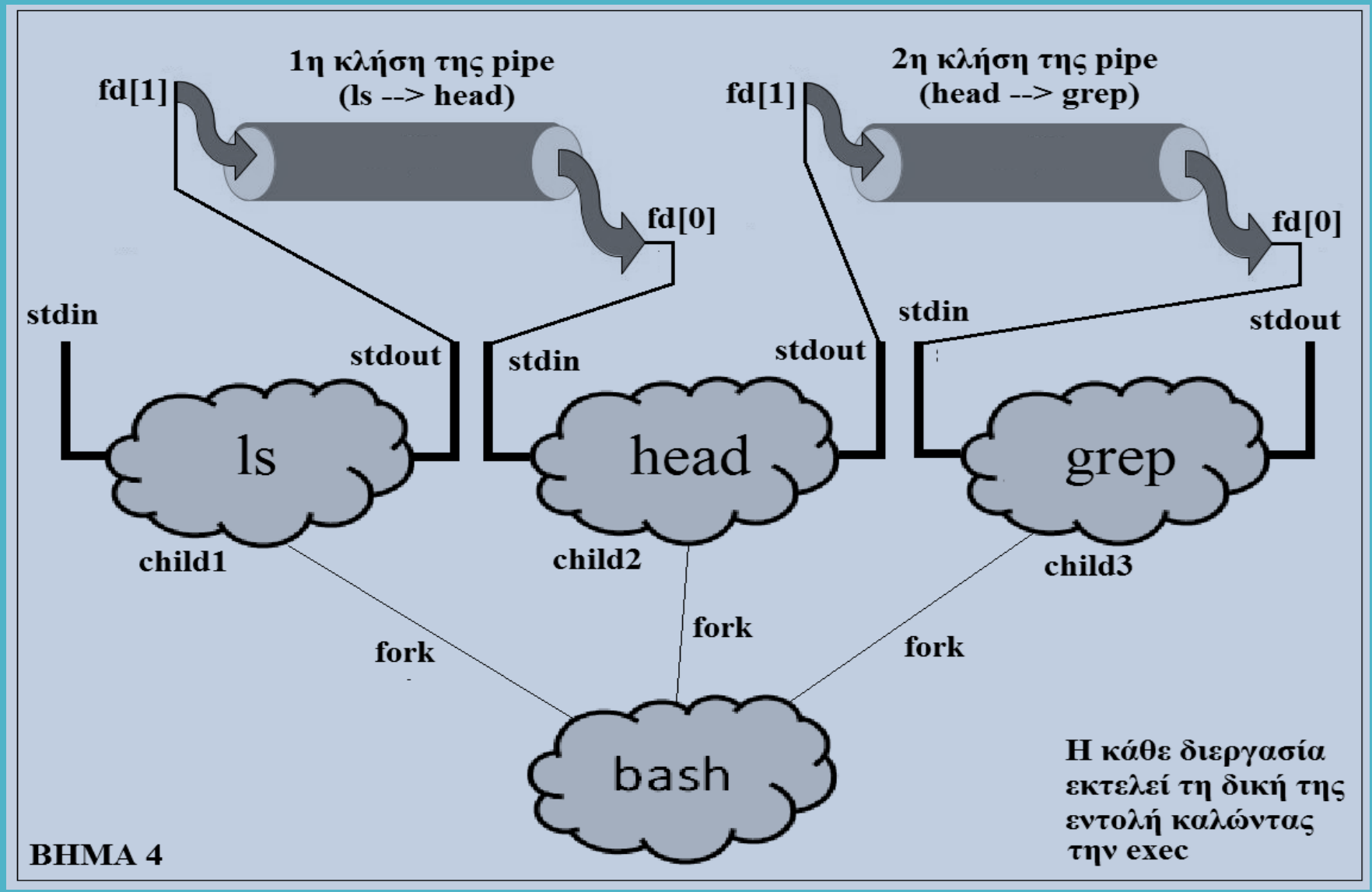
Αγωγοί

Η διαδικασία δημιουργίας και χρήσης αγωγών από το φλοιό, αποτελείται από τέσσερα βήματα:



Αγωγοί

Η διαδικασία δημιουργίας και χρήσης αγωγών από το φλοιό, αποτελείται από τέσσερα βήματα:

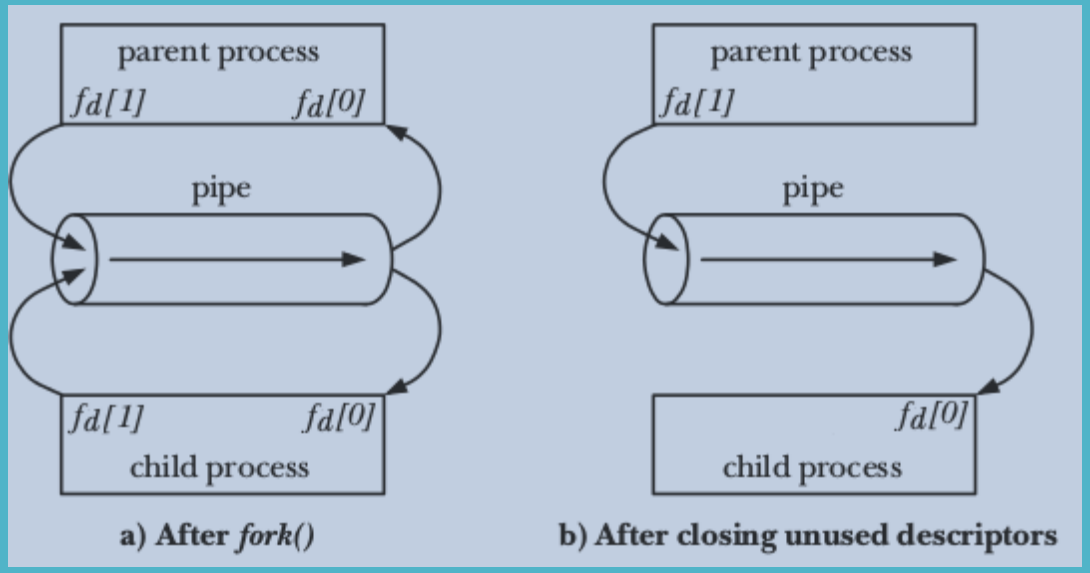


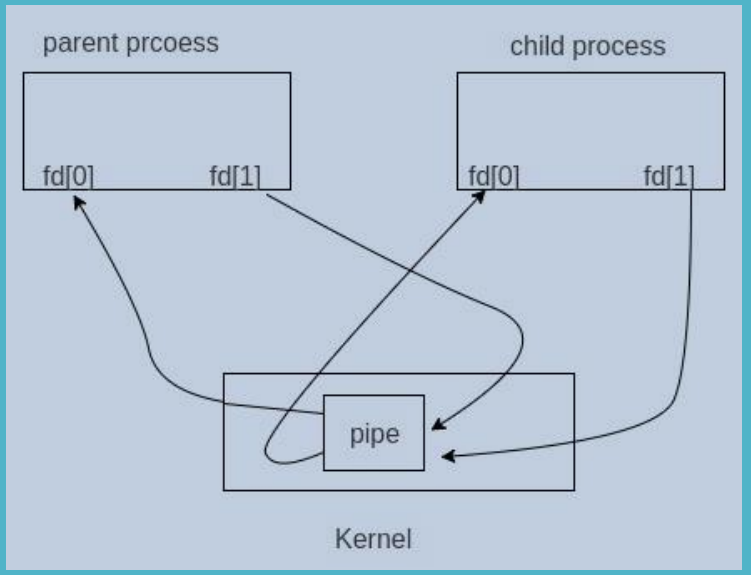
Αγωγοί

Στο παραπάνω παράδειγμα η επικοινωνία πραγματοποιείται ανάμεσα στα παιδιά της γονικής διεργασίας που βέβαια είναι ο φλοιός bash.

Η εντολή fork καλείται αυτόματα από το φλοιό χωρίς την παρέμβαση του χρήστη.

Ωστόσο είναι δυνατή η κλίση της fork μέρα από το πρόγραμμα και η δημιουργία μιας τοπολογίας πατέρα – παιδιού έτσι ώστε η επικοινωνία να μην πραγματοποιηθεί ανάμεσα στα παιδιά της γονικής διεργασίας αλλά ανάμεσα στη γονική και στη θυγατρική διεργασία.





Οι file descriptors κληροδοτούνται στη θυγατρική διεργασία και κατά συνέπεια δείχνουν στα ίδια άκρα.

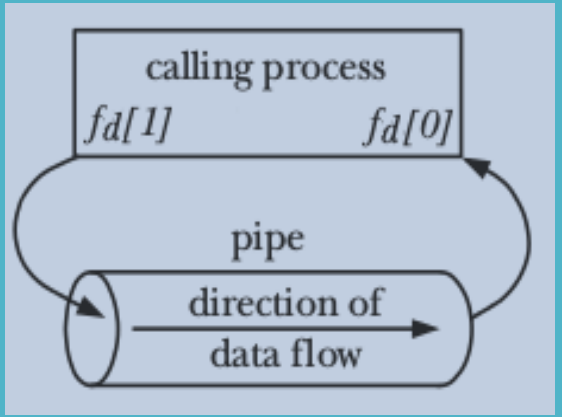
Αγωγοί

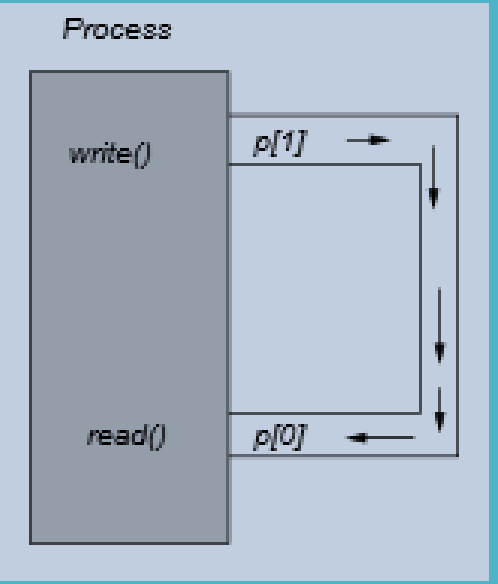
Παράδειγμα 1 → Δημιουργία και χρήση αγωγού από μία διεργασία

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

#define MSGSIZE 16


char* msg1 = "hello, world #1";
char* msg2 = "hello, world #2";
char* msg3 = "hello, world #3";

int main() {
    char inbuf[MSGSIZE];
    int p[2], i;

    if (pipe(p) < 0)
        exit(1);

    write(p[1], msg1, MSGSIZE);
    write(p[1], msg2, MSGSIZE);
    write(p[1], msg3, MSGSIZE);

    for (i = 0; i < 3; i++) {
         read(p[0], inbuf, MSGSIZE);
         printf("%s\n", inbuf); }
    return 0; }
```

```
amarg@amarg-vbox:~$ ./pipeEx1
hello, world #1
hello, world #2
hello, world #3
```

Σε αυτό το πολύ απλό παράδειγμα υπάρχει μία και μοναδική διεργασία η οποία μιλά στον εαυτό της, δηλαδή γράφει δεδομένα στο σωλήνα τα οποία στη συνέχεια διαβάζει στην είσοδό της. Δεν υπάρχει επικοινωνία μεταξύ διαφορετικών διεργασιών.



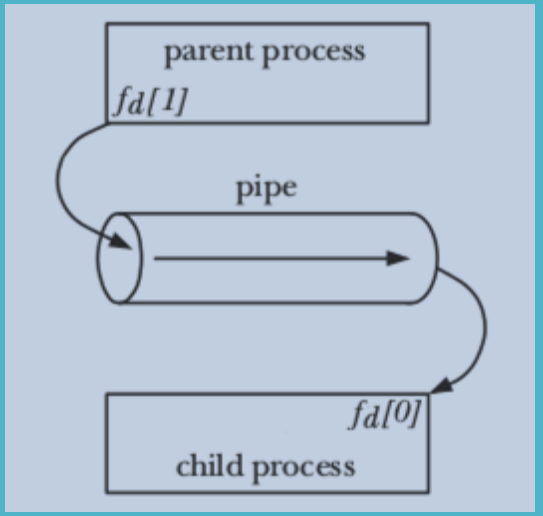


Αγωγοί

Παράδειγμα 2 → Δημιουργία και χρήση αγωγού από τοπολογία πατέρα - παιδιού

```c
int main() {
  int fd[2], errno;
  char buf;
  const char* msg = "Hello world .. Have a nice day!!\n";
  if (pipe(fd) < 0) {
     printf("Error Number : %d\n", errno);
     perror("Error Description");
     return (-1); }
  pid_t cpid = fork();
  if (cpid < 0) {
     printf("Error Number : %d\n", errno);
     perror("Error Description");
     return (-2); }
  // child process
  if (cpid==0) {
    // child only reads, so close write end
    close(fd[1]);
    while (read(fd[0], &buf, 1) > 0)
           write(STDOUT_FILENO, &buf, sizeof(buf));
    close(fd[0]);
    _exit(0); }
  // parent process
  else {
    // parent only writes, so close read end
    close(fd[0]);
    write(fd[1], msg, strlen(msg));
    close(fd[1]);
    wait(NULL);
    exit(0); }
  return 0; }
```

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




```
amarg@amarg-vbox:~$ ./pipeEx2
Hello world .. Have a nice day!!
```

Η διεργασία πατέρας γράφει στον αγωγό ολόκληρο το μήνυμα σε ένα και μόνο βήμα, ενώ η διεργασία παιδί μέσω μίας επαναληπτικής διαδικασίας διαβάζει έναν έναν τους χαρακτήρες του μηνύματος και τους εκτυπώνει στην οθόνη.

Αγωγοί

Παράδειγμα 3 → Δημιουργία και χρήση αγωγού από τοπολογία πατέρα - παιδιού

```c
int main() {
  int fd[2], errno, count;
  char buffer[1024];
  if (pipe(fd) < 0) {
     printf("Error Number : %d\n", errno);
     perror("Error Description");
     return (-1); }
  pid_t cpid = fork();
  if (cpid < 0) {
     printf("Error Number : %d\n", errno);
     perror("Error Description");
     return (-2); }
  // child process
  if (cpid==0) {
    close(fd[0]);
    printf("Child Process --> Enter an input: ");
    fgets(buffer, sizeof(buffer), stdin);
    printf("Child process: User Inpus is %s", buffer);
    count = write(fd[1], buffer, strlen(buffer)+1);
    exit(0); }
  // parent process
  else {
    close(fd[1]);
    count = read(fd[0], buffer, sizeof(buffer));
    printf("Parent process: message is %s", buffer);
    wait(NULL);   }
  return 0; }
```

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

```
amarg@amarg-vbox:~$ ./pipeEx3
Child Process --> Enter an input: Hello World !!
Child process: User Inpus is Hello World !!
Parent process: message is Hello World !!
```

Η διεργασία παιδί ζητά από το χρήστη να καταχωρήσει μία συμβολοσειρά την οποία στη συνέχεια γράφει στον αγωγό. Αυτή η συμβολοσειρά διαβάζεται στο άλλο άκρο από τη διεργασία παιδί και στη συνέχεια εκτυπώνεται στην οθόνη.

Αγωγοί

Παράδειγμα 3 → Δημιουργία και χρήση αγωγού μεταξύ θυγατρικών διεργασιών

Ο κώδικας που ακολουθεί δέχεται ως όρισμα ένα όνομα καταλόγου dirname και μία συμβολοσειρά προς αναζήτηση string και στη συνέχεια εκτελεί τη διασωλήνωση

ls –l dirname | grep string

για να εκτυπώσει τα περιεχόμενα του καταλόγου dirname που περιέχουν στο όνομά τους τη συμβολοσειρά string.

Αυτό το παράδειγμα λειτουργεί με τον ίδιο τρόπο με τον οποίο το bash shell υλοποιεί διασωληνώσεις σαν την παραπάνω τις οποίες εκτελεί ο χρήστης από τη γραμμή εντολών.

Αγωγοί

Παράδειγμα 3 → Δημιουργία και χρήση αγωγού μεταξύ θυγατρικών διεργασιών

```
int main(int argc, char *argv[]) {
    int pid, pid_ls, pid_grep, fd[2];
    if (argc != 3) {
        printf("Usage: pipeEx4 dirname string\n");
        return (-1); }
    printf("Parent process: Grepping %s for %s\n", argv[1], argv[2]);
    if (pipe(fd)==-1) {
        printf("Parent process: Failed to create pipe\n");
        return (-2); }
    // Fork a process to run grep
    pid_grep = fork();
    if (pid_grep == -1) {
        printf("Parent process: Could not fork process to run grep\n");
        return (-3); }
    else if (pid_grep==0) {
         printf("Child process 1: grep child will now run\n");
         // Set fd[0] (stdin) to the read end of the pipe
         if (dup2(fd[0], STDIN_FILENO) == -1) {
             printf("Child process 1: grep dup2 failed\n");
             return (-4); }
         // Close the pipe now that we've duplicated it
         close(fd[0]);
         close(fd[1]);
         // Setup the arguments/environment to call
         char * new_argv[] = { "/bin/grep", argv[2], 0 };
         char * envp[] = { "HOME=/", "PATH=/bin:/usr/bin", "USER=brandon", 0 };
         // Call execve(2) which will replace the executable image of this process
         execve(new_argv[0], &new_argv[0], envp);
         // Execution will never continue in this process unless execve returns
         // because of an error
         printf("Child process 1: Oops, grep failed!\n");
         return (-5); }
```

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

Αγωγοί

Παράδειγμα 3 → Δημιουργία και χρήση αγωγού μεταξύ θυγατρικών διεργασιών

```
// Fork a process to run ls
pid_ls = fork();
if (pid_ls == -1) {
    printf("Parent Process: Could not fork process to run ls\n");
    return (-6); }
else if (pid_ls == 0) {
     printf("Child process 2: ls child will now run\n");
     printf("----------------------\n");
     // Set fd[1] (stdout) to the write end of the pipe
     if (dup2(fd[1], STDOUT_FILENO) == -1) {
         fprintf(stderr, "ls dup2 failed\n");
         return (-7); }
     // Close the pipe now that we've duplicated it
     close(fd[0]);
     close(fd[1]);
     // Setup the arguments/environment to call
     char *new_argv[] = { "/bin/ls", "-la", argv[1], 0 };
     char *envp[] = { "HOME=/", "PATH=/bin:/usr/bin", "USER=brandon", 0 };
     // Call execve(2) which will replace the executable image of this process
     execve(new_argv[0], &new_argv[0], envp);
     // Execution will never continue in this process unless execve returns
     // because of an error
     fprintf(stderr, "child: Oops, ls failed!\n");
     return (-8); }
// Parent doesn't need the pipes
close(fd[0]);
close(fd[1]);
printf("Parent: Parent will now wait for children to finish execution\n");
// Wait for all children to finish
while (wait(NULL) > 0);
printf("----------------------\n");
printf("parent: Children has finished execution, parent is done\n");
return 0; }
```

Αγωγοί

Παράδειγμα 3 → Δημιουργία και χρήση αγωγού μεταξύ θυγατρικών διεργασιών

Το αποτέλεσμα της εκτέλεσης του παραπάνω κώδικα που ακολουθεί στη συνέχεια

```
amarg@amarg-vbox:~$ ./pipeEx4 shared prog
Parent process: Grepping shared for prog
Parent: Parent will now wait for children to finish execution
Child process 2: ls child will now run
--------------------
Child process 1: grep child will now run
-rwxrwxrwx  1 root  root   19624 Sep 23 17:21 prog
-rwxrwxrwx  1 root  root     407 Sep 23 17:21 prog.c
-rwxrwxrwx  1 root  root    2432 Sep 23 17:21 prog.o
--------------------
parent: Children has finished execution, parent is done
```

είναι ακριβώς το ίδιο με εκείνο που παίρνουμε εάν εκτελέσουμε στη γραμμή εντολών του λειτουργικού συστήματος την εντολή

ls –l shared | grep prog

```
amarg@amarg-vbox:~$ ls -l shared | grep prog
-rwxrwxrwx 1 root root  19624 Σεπ  23 17:21 prog
-rwxrwxrwx 1 root root    407 Σεπ  23 17:21 prog.c
-rwxrwxrwx 1 root root   2432 Σεπ  23 17:21 prog.o
```

Αυτός λοιπόν είναι και ο τρόπος με τον οποίο υλοποιούνται οι διασωληνώσεις από το φλοιό του λειτουργικού συστήματος.

Αγωγοί

Αναπαραγωγή περιγραφέων αρχείου

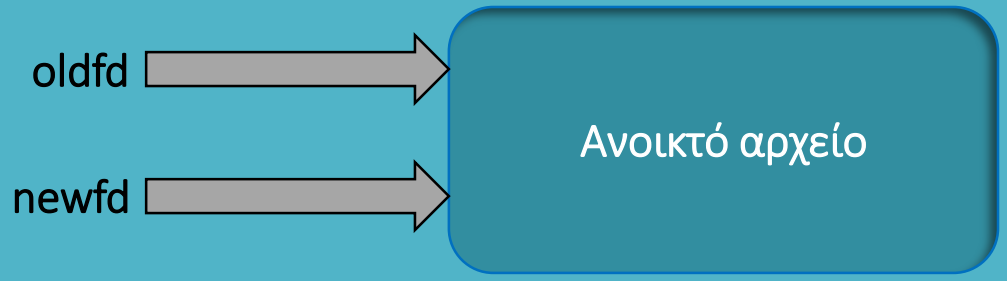
Πραγματοποιείται από τις συναρτήσεις του αρχείου unistd.h

```
int dup (int oldfd);
Int dup2 (int oldfd, int newfd);
```

οι οποίες δημιουργούν νέους περιγραφείς αρχείου που δείχνουν σε ένα ανοιχτό αρχείο. Είναι ιδιαίτερα χρήσιμες όταν ο χρήστης θέλει να ανακατευθύνει τα stdin, stdout και stderr.

Η συνάρτηση dup επιστρέφει έναν περιγραφέα αρχείου που έχει όσο το δυνατό μικρότερη τιμή, ενώ εάν θέλουμε να δημιουργήσουμε έναν περιγραφέα αρχείου με συγκεκριμένη τιμή, χρησιμοποιούμε την dup2. Σε αμφότερες τις περιπτώσεις οι περιγραφείς oldfd και newfd δείχνουν στο ίδιο αρχείο.

Εάν ο περιγραφέας newfd δείχνει σε ανοικτό αρχείο, αυτό κλείνει προκειμένου ο νέος περιγραφέας να δείξει στο ίδιο αρχείο με τον περιγραφέα oldfd.



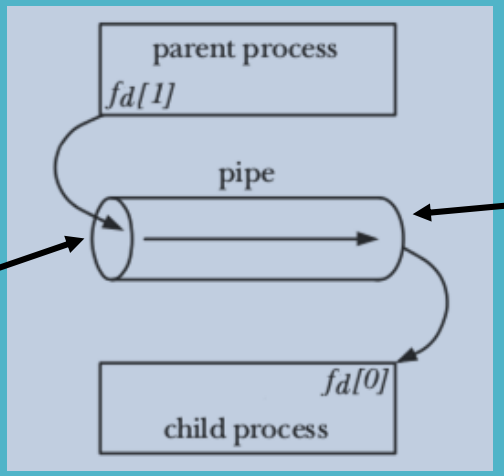
Η αναπαραγωγή ενός περιγραφέα αρχείου ΔΕΝ ΣΥΜΠΙΠΤΕΙ με το άνοιγμα του αρχείου δύο φορές.

Αγωγοί

Το βασικό χαρακτηριστικό των αγωγών που παρουσιάστηκαν μέχρι τώρα είναι πως δεν σχετίζονται με αρχεία που αποθηκεύονται στο σκληρό δίσκο. Περιγράφονται από ειδικές δομές που αποθηκεύονται προσωρινά στη μνήμη και διαγράφονται με την ολοκλήρωση της λειτουργίας.

Το μειονέκτημά τους είναι πως χρησιμοποιούνται μόνο στην περίπτωση διεργασιών που σχετίζονται μεταξύ τους με σχέση γονέα - παιδιού. Η διαδικασία έχει ως εξής:

- Η γονική διεργασία δημιουργεί τον αγωγό με την pipe.
- Η γονική διεργασία (συγγραφέας) δημιουργεί τη θυγατρική διεργασία (αναγνώστης) με τη fork.
- Η θυγατρική διεργασία κλείνει το άκρο εγγραφής του αγωγού αφού θα πραγματοποιήσει ανάγνωση.
- Η θυγατρική διεργασία εκτελεί το θυγατρικό πρόγραμμα με κάποια έκδοση της exec.
- Η γονική διεργασία κλείνει το άκρο ανάγνωσης του αγωγού αφού θα πραγματοποιήσει εγγραφή.
- Η γονική διεργασία γράφει στον αγωγό.
- Η θυγατρική διεργασία διαβάζει από τον αγωγό.



Η θυγατρική διεργασία κλείνει το άκρο εγγραφής

Η γονική διεργασία κλείνει το άκρο ανάγνωσης.

Αγωγοί

Η επικοινωνία διεργασιών που δεν σχετίζονται με σχέση γονέα – παιδιού αλλά μπορεί να είναι οποιεσδήποτε, γίνεται με τη βοήθεια των επώνυμων αγωγών που είναι γνωστοί ως FIFO.

FIFO (First In First Out)

Υφίστανται ως αρχεία στο σκληρό δίσκο και επιτρέπουν την πολλαπλή εγγραφή δεδομένων από πολλούς συγγραφείς ταυτόχρονα τα οποία διαβάζονται από έναν αναγνώστη και μέσω μιας αρχιτεκτονικής που μοιάζει με την αρχιτεκτονική client / server.

Με εξαίρεση τον διαφορετικό τρόπο δημιουργίας τους, η λειτουργία τους είναι παρόμοια με αυτή των απλών (ανώνυμων) αγωγών.

Οι επώνυμοι αγωγοί ανοίγουν με την open η οποία χρησιμοποιεί στο άκρο του αναγνώστη το flag O_RDONLY και στο άκρο του συγγραφέα το flag O_WRONLY.

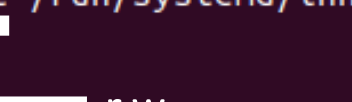
Τα αρχεία του σκληρού δίσκου που αναφέρονται σε επώνυμους αγωγούς χαρακτηρίζονται από την εμφάνιση του γράμματος p ως πρώτο γράμμα της μάσκας δικαιωμάτων που εκτυπώνει η ls –l.

```
prw------- 1 root root    0 Οκτ    3 10:59 1.ref
prw------- 1 root root    0 Οκτ    3 10:59 2.ref
```

Αγωγοί

Η εκτέλεση του προγράμματος filestat με όρισμα ένα αρχείο επώνυμου αγωγού μας δίνει

```
amarg@amarg-vbox:~/shared/Lab5$ ./filestat /run/systemd/inhibit/1.ref
File type:                FIFO/pipe
I-node number:            412
Mode:                     10600 (octal)
Link count:               1
Ownership:                UID=0   GID=0
Preferred I/O block size: 4096 bytes
File size:                0 bytes
Blocks allocated:         0
Last status change:       Sat Oct  3 10:59:23 2020
Last file access:         Sat Oct  3 10:59:23 2020
Last file modification:   Sat Oct  3 10:59:23 2020
amarg@amarg-vbox:~/shared/Lab5$
```



Η δημιουργία επώνυμων αγωγών γίνεται με τις συναρτήσεις mkfifo και mknod – από αυτές χρησιμοποιείται συνήθως η mkfifo που χαρακτηρίζεται από φορητότητα και όχι η mknod η οποία παρεμπιπτόντως εκτός από αρχεία FIFO μπορεί να κατασκευάσει και άλλους τύπους αρχείων όπως block & character device files ενώ πριν την υλοποίηση της εντολής mkdir χρησιμοποιούνταν και για τη δημιουργία καταλόγων. Η συνάρτηση mkfifo ορίζεται ως

int mkfifo (const char *pathname, mode_t mode);

Αγωγοί

Παράδειγμα δημιουργίας επώνυμου αγωγού με τη συνάρτηση mkfifo

```c
// C program to implement one side of FIFO ...
// this side writes first, then reads

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

int main() {
    int fd;

    // FIFO file path
    char * myfifo = "/tmp/myfifo";
    // Creating the named file(FIFO)
    // mkfifo(<pathname>, <permission>)
    mkfifo(myfifo, 0666);
    char arr1[80], arr2[80];
    while (1) {
        // Open FIFO for write only
        fd = open(myfifo, O_WRONLY);
        // Take an input arr2ing from user.
        // 80 is maximum length
        fgets(arr2, 80, stdin);
        // Write the input arr2ing on FIFO and close it
        write(fd, arr2, strlen(arr2)+1);
        close(fd);
        // Open FIFO for Read only
        fd = open(myfifo, O_RDONLY);
        // Read from FIFO
        read(fd, arr1, sizeof(arr1));
        // Print the read message
        printf("User2: %s\n", arr1);
        close(fd); }
    return 0; }
```

```c
// C program to implement one side of FIFO ...
// this side reads first, then writes

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

int main() {
    int fd1;

    // FIFO file path
    char * myfifo = "/tmp/myfifo";
    // Creating the named file(FIFO)
    // mkfifo(<pathname>,<permission>)
    mkfifo(myfifo, 0666);
    char str1[80], str2[80];
    while (1) {
        // First open in read only and read
        fd1 = open(myfifo,O_RDONLY);
        read(fd1, str1, 80);

        // Print the read string and close
        printf("User1: %s\n", str1);
        close(fd1);

        // Now open in write mode and write
        // string taken from user.
        fd1 = open(myfifo,O_WRONLY);
        fgets(str2, 80, stdin);
        write(fd1, str2, strlen(str2)+1);
        close(fd1); }
    return 0; }
```

Αγωγοί

Οι επώνυμοι αγωγοί περιγράφονται από πραγματικά αρχεία του συστήματος

```
amarg@amarg-vbox:~$ ls -l /tmp
total 40
-rw------- 1 amarg amarg    0 Οκτ   3 10:59 config-err-ljzuZQ
prw-rw-r-- 1 amarg amarg    0 Οκτ   3 15:38 myfifo
drwx------ 2 amarg amarg 4096 Οκτ   3 10:59 ssh-KFr1WVeZZu5b
```

Οι δύο εφαρμογές είναι ανεξάρτητες η μία από την άλλη και εκτελούνται παράλληλα σε δύο διαφορετικά τερματικά ανταλλάσσοντας μηνύματα τα οποία διακινούνται μέσω του αγωγού.