Λειτουργικά Συστήματα Υπολογιστών

Αναφορά στην 2η Εργαστηριακή Άσκηση

Αλέξανδρος Σκούρας, 03120105

Ιωάννης Τσαντήλας, 03120883

Εξάμηνο: Εαρινό 2022-23

1.1 Δημιουργία δεδομένου δέντρου διεργασιών

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

#include "proc-common.h"

#define SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3

/*
 * Create this process tree:
 * A-+-B---D
 *   `-C
 */
void fork_procs(void)
{
        pid_t pid_B, pid_C, pid_D;
        int status;

        change_pname("A");
        printf("Creating A...\n");
        pid_B=fork();

        if(pid_B < 0){
            perror("fork");
            exit(1);
        }

        if(pid_B == 0){  //Process B
                change_pname("B");
                printf("Creating B...\n");
                pid_D=fork();

                if(pid_D < 0){ //Error check
                        perror("fork");
                        exit(1);
                }

                if(pid_D == 0){ //Process D
                        change_pname("D");

                        printf("Creating D...\n");
                        printf("D Sleeping...\n");

                        sleep(SLEEP_PROC_SEC);
                        printf("D Exiting...\n");
                        exit(13);
                }

                if(pid_D > 0){ //B waits D to terminate
                        printf("B Waiting...\n");

                        pid_D=wait(&status);
                        explain_wait_status(pid_D, status);

                        printf("B Exiting...\n");
                        exit(19);
                }
        }

        if(pid_B > 0){
                pid_C=fork();
```

```
                if(pid_C < 0){ //Error check
                        perror("fork");
                        exit(1);
                }

                if(pid_C == 0){ //Process C
                        change_pname("C");

                        printf("Creating C...\n");
                        printf("C Sleeping...\n");

                        sleep(SLEEP_PROC_SEC);
                        printf("C Exiting...\n");
                        exit(17);
                }

                if(pid_C > 0){ //A waits B and C to terminate
                        printf("A Waiting...\n");

                        pid_B=wait(&status);
                        explain_wait_status(pid_B, status);

                        pid_C=wait(&status);
                        explain_wait_status(pid_C, status);

                        printf("A Exiting...\n");
                        exit(16);
                }

        }
}

int main(void) {
        pid_t pid_A;
        int status;

        /* Fork root of process tree */
        pid_A = fork();

        if (pid_A < 0) {
                perror("main: fork");
                exit(1);
        }

        if (pid_A == 0) {
                fork_procs();
                exit(1);
        }

        sleep(SLEEP_TREE_SEC);

        /* Print the process tree root at pid */
        show_pstree(pid_A);

        /* Wait for the root of the process tree to terminate */
        pid_A = wait(&status);
        explain_wait_status(pid_A, status);

        return 0;
}
```

Η έξοδος του προγράμματος:

```
Creating A...
A Waiting...
Creating C...
Creating B...
C Sleeping...
B Waiting...
Creating D...
D Sleeping...


A(29604)─┬─B(29605)──D(29607)
         └─C(29606)


C Exiting...
My PID = 29604: Child PID = 29606 terminated normally, exit status = 17
D Exiting...
My PID = 29605: Child PID = 29607 terminated normally, exit status = 13
B Exiting...
My PID = 29604: Child PID = 29605 terminated normally, exit status = 19
A Exiting...
My PID = 29603: Child PID = 29604 terminated normally, exit status = 16
```

Σημείωση

Η συνάρτηση **wait()**είναι ένα system call που αναστέλλει την λειτουργία μιας διεργασίας έως ότου κάποιο από τα παιδιά της τερματιστεί. Όταν συμβεί αυτό, το λειτουργικό στέλνει το κατάλληλο σήμα ώστε ο πατέρας να συνεχίσει (για αυτό, εάν ο πατέρας έχει πολλαπλά παιδιά, καλούμε την **wait**πολλαπλές φορές).

Ερωτήσεις

1. Εάν σκοτώσουμε πρόωρα την διαδικασία A με **kill -KILL <pid>**, τότε όλα τα παιδιά-διαδικασίες της A (δηλαδή οι B,C) θα γίνουν ορφανές διαδικασίες, οι οποίες θα αποκτήσουν ως νέο πατέρα την init διαδικασία, με PID=1.
2. Εάν γράψουμε **show_pstree(getpid())** αντί για **show_pstree(pid)**η έξοδος του προγράμματος είναι:

```
test1(5987)─┬─A(5988)─┬─B(5989)──D(5991)
            │         └─C(5990)
            └─sh(5995)──pstree(5996)
```

Όπου**test1**είναι το εκτελέσιμο του προγράμματος, **sh**και **pstree**είναι διαδικασίες πουπροκύπτουν από την **show_pstree()**. Με την χρήση του **pid**, καλούμε την συνάρτηση

για την διαδικασία Α, ενώ με την **getpid()**την καλούμε για το εκτελέσιμο, εξ’ ου και οι επιπλέον διαδικασίες.

3. Αυτό μπορεί να συμβαίνει επειδή οι **διαδικασίες καταναλώνουν πόρους** του συστήματος, όπως CPU time, μνήμη και δίσκο Ε/Ε, επομένως πολλές διαδικασίες μπορούν να μειώσουν την αποδοτικότητα του συστήματος. Άλλοι λόγοι είναι η **ασφάλεια** περιορίζοντας το πλήθος των πόρων που οι κακόβουλοι hackers θα μπορέσουν να αξιοποιήσουν και **δικαιοσύνη**, ώστε κάθε χρήστης να έχει ισάξιο πλήθος πόρων.

1.2 Δημιουργία αυθαίρετου δέντρου διαδικασιών

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

#include "proc-common.h"
#include "tree.h"

#define SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3


void fork_procs(struct tree_node *root) {
        int status;
        pid_t child;
        change_pname(root->name);
        printf("Creating %s...\n", root->name);
        if(root->nr_children == 0){//Leaf node
                printf("%s Sleeping...\n", root->name);

                sleep(SLEEP_PROC_SEC);

                printf("%s Exiting...\n", root->name);
                exit(17);//Exit code for leaf processes
        }

        //Non-leaf process
        for(int i=0; i<root->nr_children; i++){//Create children processes
                child=fork();

                if(child < 0){//Error check
                        perror("fork");
                        exit(1);
                }

                if(child == 0){//Child Process
                        fork_procs(root->children+i);
                }
        }
        //Father
        printf("%s Waiting...\n", root->name);

        for(int i=0; i<root->nr_children; i++){//Wait for all children to terminate
               child=wait(&status);
               explain_wait_status(child, status);
        }

        printf("%s Exiting...\n", root->name);
               exit(42);//Exit code for non-leaf processes
}

int main(int argc, char **argv) {
        pid_t pid;
        int status;
        struct tree_node *root;

        if(argc!=2){//Check arguments
                fprintf(stderr, "Usage: %s <input_tree_file>\n\n", argv[0]);
                exit(1);
        }

        root=get_tree_from_file(argv[1]);
        print_tree(root);

        /* Fork root of process tree */
        pid = fork();
        if (pid < 0) {//Error check
                perror("main: fork");
                exit(1);
        }
```

```c
        }

        if (pid == 0) {//Child
                fork_procs(root);
                exit(1);
        }
        //Father
        sleep(SLEEP_TREE_SEC);//Sleep until process tree is created

        /* Print the process tree root at pid */
        show_pstree(pid);


        /* Wait for the root of the process tree to terminate */
        pid = wait(&status);
        explain_wait_status(pid, status);

        return 0;
}
```

Για το εξής αρχείο εισόδου:

```
# file that defines the tree
# lines starting with '#' are comments
# . each block of lines defines a node
# . each node is defined as:
#    1st line:          name of node
#    2nd line:          number of children
#    subsequent lines: name(s) of children
# . blocks are seperated with empty lines
# . no comments are allowed within a block
# . nodes must be placed in a DFS order

A
3
B
C
D

B
2
E
F

E
0

F
0

C
0

D
1
G

G
0
```

Έχουμε έξοδο:

```
A
        B
                E
                F
        C
        D
                G
Creating A...
A Waiting...
Creating D...
D Waiting...
Creating B...
B Waiting...
Creating C...
C Sleeping...
Creating G...
G Sleeping...
Creating E...
E Sleeping...
Creating F...
F Sleeping...


A(30568)──┬─B(30569)──┬─E(30573)
          │           └─F(30574)
          ├─C(30570)
          └─D(30571)──G(30572)
C Exiting...
My PID = 30568: Child PID = 30570 terminated normally, exit status = 17
G Exiting...
My PID = 30571: Child PID = 30572 terminated normally, exit status = 17
D Exiting...
E Exiting...
My PID = 30568: Child PID = 30571 terminated normally, exit status = 42
My PID = 30569: Child PID = 30573 terminated normally, exit status = 17
F Exiting...
My PID = 30569: Child PID = 30574 terminated normally, exit status = 17
B Exiting...
My PID = 30568: Child PID = 30569 terminated normally, exit status = 42
A Exiting...
My PID = 30567: Child PID = 30568 terminated normally, exit status = 42
```

Ερωτήσεις

1. Τα μηνύματα «εισόδου» (starting) τυπώνονται με βάση την BFS διάσχιση του δέντρου(Βέβαια και εδώ μπορεί να υπάρξει τυχαιότητα). Τα μηνύματα «εξόδου» (exiting) δεν έχουν την ίδια σειρά με αυτά της εισόδου ή με την σειρά που εκτελούνται. Η σειρά εξαρτάται από τον αλγόριθμο δρομολόγησης του

λειτουργικού, τον εκτελούμενο χρόνο κάθε διαδικασίας ή το πλήθος των διαδικασιών που συναγωνίζονται για τους πόρους. Επιπλέον, εάν ο πατέρας περιμένει πολλά παιδιά να τερματιστούν, η σειρά τερματισμού δεν είναι απαραίτητα προκαθορισμένη.

1.3 Αποστολή και χειρισμός σημάτων

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

#include "tree.h"
#include "proc-common.h"

extern int kill(pid_t pid, int sig);

void fork_procs(struct tree_node *root)
{
        int status;

        change_pname(root->name);
        printf("PID = %ld, name %s, starting...\n", (long)getpid(), root->name);

        if(root->nr_children == 0) {//Leaf node
                raise(SIGSTOP);
                printf("PID = %ld, name = %s is awake\n", (long)getpid(), root->name);
                exit(17);//Exit code for leaf processes
        }

        else{//Non-leaf node

                pid_t child[root->nr_children];//Create table for child pid to remembe
r our children to use SIGCONT later
                for(int i=0; i<root->nr_children; ++i){//Create children processes
                        child[i]=fork();

                        if(child[i]<0){//Error check
                                perror("fork");
                                exit(1);
                        }

                        if(child[i] == 0){//Child
                                fork_procs(root->children+i);
                        }
                }
                //Father
                wait_for_ready_children(root->nr_children);//Wait for all children to
stop

                raise(SIGSTOP);

                printf("PID = %ld, name = %s is awake\n", (long)getpid(), root->name);

                for(int i=0; i<root->nr_children; i++){

                        kill(child[i], SIGCONT);//Wake up all children

                        wait(&status);

                        explain_wait_status(child[i], status);
                }

                        exit(42);//Exit code for non leaf processes


        }
}
```

```c
int main(int argc, char *argv[]) {
        pid_t pid;
        int status;
        struct tree_node *root;

        if (argc != 2){//Check arguments
                fprintf(stderr, "Usage: %s <tree_file>\n", argv[0]);
                exit(1);
        }

        /* Read tree into memory */
        root = get_tree_from_file(argv[1]);

        /* Fork root of process tree */
        pid = fork();

        if (pid < 0) {//Error check
                perror("main: fork");
                exit(1);
        }

        if (pid == 0) {
                /* Child */
                fork_procs(root);
                exit(1);
        }

        /*
         * Father
         */

        wait_for_ready_children(1);//Wait for the root to stop

        /* Print the process tree root at pid */
        show_pstree(pid);


        kill(pid, SIGCONT);//Root wakes up

        /* Wait for the root of the process tree to terminate */
        wait(&status);
        explain_wait_status(pid, status);

        return 0;
}
```

Για το ίδιο αρχείο εισόδου με το ερώτημα 1.2, έχουμε έξοδο:

```
PID = 20477, name A, starting...
PID = 20480, name D, starting...
PID = 20478, name B, starting...
PID = 20481, name G, starting...
My PID = 20480: Child PID = 20481 has been stopped by a signal, signo = 19
My PID = 20477: Child PID = 20480 has been stopped by a signal, signo = 19
PID = 20482, name E, starting...
My PID = 20478: Child PID = 20482 has been stopped by a signal, signo = 19
PID = 20479, name C, starting...
My PID = 20477: Child PID = 20479 has been stopped by a signal, signo = 19
PID = 20483, name F, starting...
My PID = 20478: Child PID = 20483 has been stopped by a signal, signo = 19
My PID = 20477: Child PID = 20478 has been stopped by a signal, signo = 19
My PID = 20476: Child PID = 20477 has been stopped by a signal, signo = 19


A(20477)─┬─B(20478)─┬─E(20482)
         │          └─F(20483)
         ├─C(20479)
         └─D(20480)──G(20481)


PID = 20477, name = A is awake
PID = 20478, name = B is awake
PID = 20482, name = E is awake
My PID = 20478: Child PID = 20482 terminated normally, exit status = 17
PID = 20483, name = F is awake
My PID = 20478: Child PID = 20483 terminated normally, exit status = 17
My PID = 20477: Child PID = 20478 terminated normally, exit status = 42
PID = 20479, name = C is awake
My PID = 20477: Child PID = 20479 terminated normally, exit status = 17
PID = 20480, name = D is awake
PID = 20481, name = G is awake
My PID = 20480: Child PID = 20481 terminated normally, exit status = 17
My PID = 20477: Child PID = 20480 terminated normally, exit status = 42
My PID = 20476: Child PID = 20477 terminated normally, exit status = 42
```

Ερωτήσεις

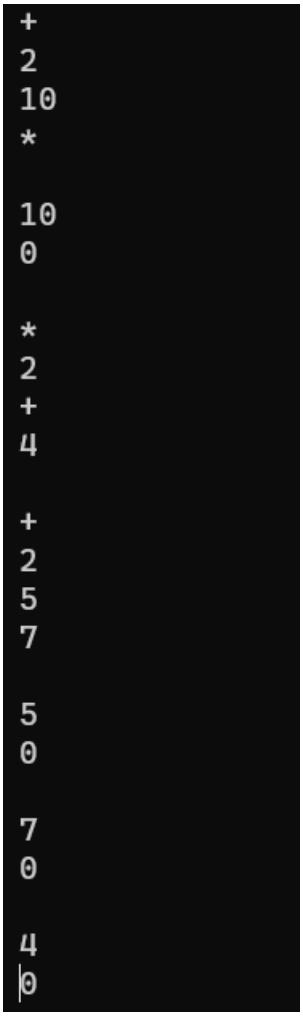
1. Τα πλεονεκτήματα της χρήσης σημάτων ενάντια της **sleep()** είναι η **αμεσότητα** και την **ακρίβεια** που προσφέρουν στον χειρισμό των διαδικασιών, όπου σε αντίθεση με την **sleep()**, μπορούμε να ελέγξουμε ακριβώς πότε θα σταματήσει/συνεχίσει, προσφέροντας μας **ασύγχρονη λειτουργία** του προγράμματος μας.
2. Η **wait_for_ready_children** περιμένει όλα τα παιδιά της διεργασίας που την κάλεσε να σταματήσουν. Την χρειαζόμαστε έτσι ώστε να εξασφαλίσουμε πως κάθε παιδί-διεργασία έχει δημιουργηθεί και είναι έτοιμο να εκτελεσθεί πριν από τον πατέρα-διεργασία, διαφορετικά θα προκύψουν προβλήματα συγχρονισμού, όπως π.χ. η αλληλεπίδραση του πατέρα με ένα μη έτοιμο παιδί.

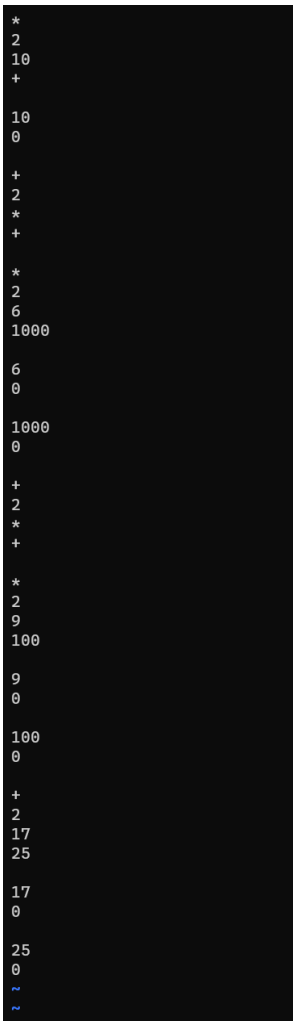
1.4 Παράλληλος υπολογισμός αριθμητικής έκφρασης

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

#include "proc-common.h"
#include "tree.h"

#define SLEEP_PROC_SEC  5


void fork_procs(struct tree_node *root, int fd) {
        int status;
        pid_t child;

        change_pname(root->name);
        printf("Creating %s...\n", root->name);

        if(root->nr_children==0){//Leaf node
                int val = atoi(root->name);//Convert string to int

                if(write(fd, &val, sizeof(val))!= sizeof(val)){//Write value of leaf p
rocess to pipe
                        perror("Write");
                        exit(1);
                }
                printf("%s Sleeping...\n", root->name);
                sleep(SLEEP_PROC_SEC);
                printf("%s Exiting...\n", root->name);
                exit(17);//Exit code for leaf processes
        }
       else{//Non-leaf code
        int k, pfd[2];
        k=pipe(pfd);//Create pipe then check for error

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

        for(int i=0; i<root->nr_children; i++){//Create children processes
                child=fork();

                if(child < 0){//Error check
                        perror("fork");
                        exit(1);
                }

                if(child == 0){//Child
                        fork_procs(root->children+i, pfd[1]);
                }
        }
        //Father
        printf("%s Waiting...\n", root->name);

        int value[2];

        for(int i=0; i<root->nr_children; i++){
                if(read(pfd[0], &value[i], sizeof(value[i]))!= sizeof(value[i])){//Rea
d values of both children then do the correct operation
                        perror("read");
                        exit(1);
                }
        }
```

```c
        int mid_res;

        if(strcmp(root->name, "+") == 0){//+ Operation
                mid_res = value[0]+value[1];
                printf("The result of %i + %i is: %i\n", value[0], value[1], mid_res);
        }

        else{//* Operation
                mid_res = value[0]*value[1];
                printf("The result of %i * %i is: %i\n", value[0], value[1], mid_res);
        }

        if(write(fd, &mid_res, sizeof(mid_res))!= sizeof(mid_res)){//Write mid result
to pipe
                 perror("Write");
                 exit(1);
        }

        for(int i=0; i<root->nr_children; i++){//Wait for all children to terminate
                wait(&status);
                explain_wait_status(child, status);
        }

        printf("%s Exiting...\n", root->name);
        exit(42);//Exit code for non-leaf processes
       }
}

int main(int argc, char **argv) {
        pid_t pid;
        int status, k, result, pfd[2];
        struct tree_node *root;

        if(argc!=2){//Check arguments
                fprintf(stderr, "Usage: %s <input_tree_file>\n\n", argv[0]);
                exit(1);
        }

        k=pipe(pfd);//Create pipe then check for error
        if(k < 0){
                perror("main:pipe");
                exit(1);}


        root=get_tree_from_file(argv[1]);
        print_tree(root);

        /* Fork root of process tree */
        pid = fork();
        if (pid < 0) {//Error check
                perror("main: fork");
                exit(1);
        }

        if (pid == 0) {//Child
                fork_procs(root, pfd[1]);
        }
        //Father
        if(read(pfd[0], &result, sizeof(result))!= sizeof(result)){//Read final result
 from pipe(waits until something is written)
                perror("read");
                exit(1);
        }

        show_pstree(pid);
        /* Wait for the root of the process tree to terminate */
        pid = wait(&status);
        explain_wait_status(pid, status);
        printf("Final result: %i\n", result);
        return 0;
}
```

Για τα παρακάτω αρχεία εισόδου, έχουμε αντίστοιχα:





```
+
        10
        *
                +
                        5
                        7
                4
Creating +...
+ Waiting...
Creating 10...
10 Sleeping...
Creating *...
* Waiting...
Creating +...
+ Waiting...
Creating 4...
4 Sleeping...
Creating 5...
5 Sleeping...
Creating 7...
The result of 5 + 7 is: 12
The result of 4 * 12 is: 48
The result of 10 + 48 is: 58
7 Sleeping...


+(17503)─┬─*(17505)─┬─+(17506)─┬─5(17508)
         │          │          └─7(17509)
         │          └─4(17507)
         └─10(17504)


10 Exiting...
My PID = 17503: Child PID = 17505 terminated normally, exit status = 17
4 Exiting...
My PID = 17505: Child PID = 17507 terminated normally, exit status = 17
5 Exiting...
My PID = 17506: Child PID = 17509 terminated normally, exit status = 17
7 Exiting...
My PID = 17506: Child PID = 17509 terminated normally, exit status = 17
+ Exiting...
My PID = 17505: Child PID = 17507 terminated normally, exit status = 42
* Exiting...
My PID = 17503: Child PID = 17505 terminated normally, exit status = 42
+ Exiting...
My PID = 17502: Child PID = 17503 terminated normally, exit status = 42
Final result: 58
```

```
*
        10
        +
                *
                        6
                        1000
                +
                        *
                                9
                                100
                        +
                                17
                                25
Creating *...
* Waiting...
Creating 10...
10 Sleeping...
Creating +...
+ Waiting...
Creating *...
* Waiting...
Creating +...
+ Waiting...
Creating +...
+ Waiting...
Creating 1000...
1000 Sleeping...
Creating *...
* Waiting...
Creating 6...
6 Sleeping...
The result of 1000 * 6 is: 6000
Creating 25...
25 Sleeping...
Creating 17...
The result of 25 + 17 is: 42
17 Sleeping...
Creating 9...
9 Sleeping...
Creating 100...
100 Sleeping...
The result of 9 * 100 is: 900
The result of 42 + 900 is: 942
The result of 6000 + 942 is: 6942
The result of 10 * 6942 is: 69420


*(17530)──┬──+(17532)──┬──*(17533)──┬──1000(17536)
          │            │            └──6(17535)
          │            └──+(17534)──┬──*(17537)──┬──100(17542)
          │                         │            └──9(17541)
          │                         └──+(17538)──┬──17(17539)
          │                                      └──25(17540)
          └──10(17531)
```

```
10 Exiting...
My PID = 17530: Child PID = 17532 terminated normally, exit status = 17
1000 Exiting...
My PID = 17533: Child PID = 17536 terminated normally, exit status = 17
6 Exiting...
My PID = 17533: Child PID = 17536 terminated normally, exit status = 17
* Exiting...
My PID = 17532: Child PID = 17534 terminated normally, exit status = 42
25 Exiting...
My PID = 17538: Child PID = 17540 terminated normally, exit status = 17
17 Exiting...
My PID = 17538: Child PID = 17540 terminated normally, exit status = 17
+ Exiting...
9 Exiting...
My PID = 17534: Child PID = 17538 terminated normally, exit status = 42
My PID = 17537: Child PID = 17542 terminated normally, exit status = 17
100 Exiting...
My PID = 17537: Child PID = 17542 terminated normally, exit status = 17
* Exiting...
My PID = 17534: Child PID = 17538 terminated normally, exit status = 42
+ Exiting...
My PID = 17532: Child PID = 17534 terminated normally, exit status = 42
+ Exiting...
My PID = 17530: Child PID = 17532 terminated normally, exit status = 42
* Exiting...
My PID = 17529: Child PID = 17530 terminated normally, exit status = 42
Final result: 69420
```

Ερωτήσεις

1. Στην συγκεκριμένη περίπτωση, και επειδή οι πράξεις της **πρόσθεσης** και του **πολλαπλασιασμού** είναι αντιμεταθετικές, χρειαζόμαστε και χρησιμοποιούμε **μία** μόνο σωλήνωση για την επικοινωνία της κάθε διεργασίας γονέα με τα παιδιά του(Δεν μας ενδιαφέρει η σειρά επιστροφής των αποτελεσμάτων στον γονέα). Σε αντίθετη περίπτωση, στις πράξεις της **αφαίρεσης**, **ακέραιας διαίρεσης** και **ακέραιου υπόλοιπου**, οι οποίες δεν είναι αντιμεταθετικές, θα χρειαστούμε **δύο** σωληνώσεις για την επικοινωνία της κάθε διεργασίας γονέα με τα παιδιά του(Μας ενδιαφέρει η σειρά επιστροφής των αποτελεσμάτων στον γονέα).
2. Το βασικό πλεονέκτημα είναι η απόδοση. Υπολογίζοντας παράλληλα πολλαπλές διεργασίες, ο συνολικός χρόνος εκτέλεσης είναι μικρότερος από αυτόν που θα χρειαζόμασταν εάν υπολογίζαμε κάθε διεργασία ξεχωριστά.