Λειτουργικά Συστήματα Υπολογιστών

Κακούρης Δημήτριος (03119019)
Μαρδίκης Κωνσταντίνος (03119867)

2η Εργαστηριακή Άσκηση

Makefile 2ης σειράς:

Καλούμε κάθε φορά το κατάλληλο make (filename) ώστε να κάνουμε τη μεταγλώττιση και το linking.
Έχουμε κρατήσει τα ονόματα των αρχείων .c που δόθηκαν και τα επεκτείναμε. Συγκεκριμένα τα executables αντιστοιχίζονται στις ασκήσεις με τον εξής τρόπο:

1.1 -> ask2-fork
1.2 -> tree-example
1.3 -> ask2-signals
1.4 -> pipe-example

```
.PHONY: all clean

all: fork-example tree-example ask2-fork ask2-signals pipe-example

CC = gcc
CFLAGS = -g -Wall -O2
SHELL= /bin/bash

tree-example: tree-example.o tree.o proc-common.o
	$(CC) $(CFLAGS) $^ -o $@

fork-example: fork-example.o proc-common.o
	$(CC) $(CFLAGS) $^ -o $@

ask2-fork: ask2-fork.o proc-common.o
	$(CC) $(CFLAGS) $^ -o $@
```

```
ask2-signals: ask2-signals.o proc-common.o tree.o
	$(CC) $(CFLAGS) $^ -o $@

pipe-example: pipe-example.o proc-common.o tree.o
	$(CC) $(CFLAGS) $^ -o $@
%.s: %.c
	$(CC) $(CFLAGS) -S -fverbose-asm $<

%.o: %.c
	$(CC) $(CFLAGS) -c $<

%.i: %.c
	gcc -Wall -E $< | indent -kr > $@

clean:
	rm -f *.o tree-example fork-example pstree-this
ask2-{fork,tree,signals,pipes}
```

1.1 Δημιουργία δεδομένου δέντρου διεργασιών

Πηγαίος Κώδικας:

```
#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

/*
```

```c
* Create this process tree:
* A-+-B---D
*   `-C
*/
void fork_procs(void)
{
   /*
    * initial process is A.
    */

   int status;
   change_pname("A");

   pid_t pidC,pidB,pidD;


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

       change_pname("B");
       printf("B: Waiting...\n"); /* Wait for the children to be created*/



       pidD = fork();
       if (pidD < 0) {
           perror("main: fork");
           exit(1);
       }
       if (pidD== 0) {
           /* Child */
           change_pname("D");
           printf("D: Sleeping...\n");
           sleep(SLEEP_PROC_SEC); /* Sleep so the tree can be printed*/

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

       pidD = wait(&status);
       explain_wait_status(pidD, status);
       printf("B: Exiting...\n");
       exit(19);
   }
```

```c
    pidC = fork();
    if (pidC < 0) {
        perror("main: fork");
        exit(1);
    }
    if (pidC== 0) {

        change_pname("C");
        printf("C: Sleeping...\n");
        sleep(SLEEP_PROC_SEC); /* Sleep so the tree can be printed*/
        printf("C: Exiting...\n");
        exit(17);
    }
    pidC = wait(&status);
    explain_wait_status(pidC, status);


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




    /* ... */

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

/*
* The initial process forks the root of the process tree,
* waits for the process tree to be completely created,
* then takes a photo of it using show_pstree().
*
* How to wait for the process tree to be ready?
* In ask2-{fork, tree}:
*      wait for a few seconds, hope for the best.
* In ask2-signals:
*      use wait_for_ready_children() to wait until
*      the first process raises SIGSTOP.
*/
int main(void)
{
    pid_t pidA;
    int status;

    /* Fork root of process tree */
    pidA = fork();
    change_pname("A");
    if (pidA < 0) {
        perror("main: fork");
```

```
        exit(1);
    }
    if (pidA== 0) {
        printf("A: Waiting...\n");
        /* Child */
        fork_procs();

    }


    /*
     * Father
     */
    /* for ask2-signals */
    /* wait_for_ready_children(1); */

    /* for ask2-{fork, tree} */
    sleep(SLEEP_TREE_SEC);

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

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


    /* for ask2-signals */
    /* kill(pid, SIGCONT); */

    /* Wait for the root of the process tree to terminate */

    return 0;
}
```

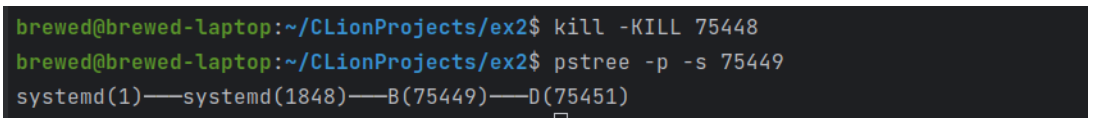
Τρέχουμε το process tree με `./ask2-fork` έχοντας τρέξει `make ask2-fork`

```
brewed@brewed-laptop:~/CLionProjects/ex2/ex2$ ./ask2-fork
A: Waiting...
B: Waiting...
C: Sleeping...
D: Sleeping...


A(73696)─┬─B(73697)──D(73699)
         └─C(73698)


D: Exiting...
C: Exiting...
My PID = 73696: Child PID = 73698 terminated normally, exit status = 17
My PID = 73697: Child PID = 73699 terminated normally, exit status = 13
B: Exiting...
My PID = 73696: Child PID = 73697 terminated normally, exit status = 19
A: Exiting...
My PID = 73695: Child PID = 73696 terminated normally, exit status = 16
```

- Ερωτήσεις:

1. Σύμφωνα με τη θεωρία σκοτώνοντας τη ρίζα του δέντρου τα παιδιά “υιοθετούνται” από την init διεργασία του λειτουργικού συστήματος. Χρειάστηκε χειροκίνητα να μεγαλώσουμε το SLEEP_PROC_SEC ώστε να προλάβουμε να εκτελέσουμε τα παρακάτω.

Η διεργασία ρίζα σκοτώθηκε από παράλληλο terminal:

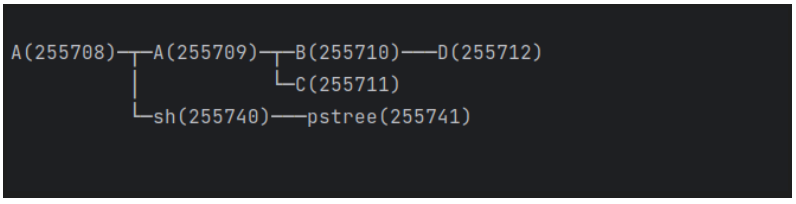


Όπως βλέπουμε όμως τυπώνοντας μετα το kill signal το δέντρο διεργασιών στο κλαδί της B(75449) υιοθετήθηκε οχι απο την init , δηλαδή systemd(1) , αλλά απο την systemd(1848).

Systemd, besides having the init process (PID 1), also creates user-level systemd instances with a non-zero PID. These instances manage user processes and services in a per-user basis, separated from the system-wide processes managed by PID 1.

Άρα υιοθετήθηκε από ένα παιδί της init το οποίο αντιστοιχεί σε ενός είδους per-user instance.

2. Με αντικατάσταση του `show_pstree(pid)` με `show_pstree(getpid())` εμφανίζεται το εξής δέντρο διεργασιών:



Οι παραπάνω διεργασίες είναι οι pstree, sh και μία παραπάνω A (ρίζα) διεργασία. Η διεργασία show_pstree εμφανίζεται ως παιδί της διεργασίας από την οποία κάνει branch out και το δέντρο, η εμφάνιση της είναι αναμενόμενη καθώς την ώρα που τυπώνεται το δέντρο, η διεργασία εκτελείται. Η διεργασία sh είναι το αποτέλεσμα όταν εκτελούμε ένα πρόγραμμα σε shell περιβάλλον και εμφανίζεται και αυτή καθώς την ώρα που τυπώνεται το δέντρο εκτελείται ο κώδικας απο shell περιβάλλον.

3. Σε υπολογιστικά συστήματα ο διαχειριστής θέτει όριο διεργασιών για κάθε χρήστη για να αποφευχθεί η εκ παραδρομής ή και κακόβουλη δημιουργία υπερβολικού αριθμού διεργασιών με αποτέλεσμα την κατάχρηση των πόρων του συστήματος, ακόμα και τη κατάρρευση της λειτουργίας του συστήματος. Παράδειγμα κακόβουλης δημιουργίας: fork bomb εκθετικά αυξανόμενος αριθμός διεργασιών χωρίς τερματισμό προκειμένου να καταρρεύσει το υπολογιστικό σύστημα.

1.2 Δημιουργία αυθαίρετου δέντρου διεργασιών

Πηγαίος κώδικας:

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

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

void fproc(struct tree_node *root) {
   int status,i;
   change_pname(root->name);
   printf("%s: Starting...\n", root->name);

   if (root->nr_children == 0) {
       printf("%s: Sleeping...\n", root->name);
       printf("%s: Exiting...\n", root->name);
       sleep(3); /* Sleep so the tree can be printed */
       exit(8); /* Exit code chosen for leaf nodes */
   } else {
       pid_t pid;
       for (i=0; i<root->nr_children; i++){
           pid=fork();
           if (pid < 0) {
               perror("main: fork");
               exit(1);
```

```
            }
            if (pid == 0) {
                fproc(root->children + i);
                exit(10);
            }
        }

    for(i=0; i<root->nr_children; i++) {
        pid = wait(&status);
        explain_wait_status(pid, status);
    } /* Loop through every child and get their status*/

    }
}

int main(int argc, char *argv[])
{
    struct tree_node *root;


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

    root = get_tree_from_file(argv[1]); /* Get Tree expression */
    print_tree(root);

    pid_t initial;
    int status;
    initial=fork();

    if (initial < 0) {
        perror("main: fork");
        exit(1);
    }
    if(initial==0){
        fproc(root);
        exit(10);
    }

    show_pstree(initial);
    sleep(10);

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


    return 0;
}
```

Κάνουμε αρχικά `make tree-example` και στη συνέχεια εκτελούμε με όρισμα το proc.tree

```
./tree-example proc.tree
```

1η εκτέλεση :

```
oslab122@orion:~/ex2$ ./tree-example proc.tree
A
        B
                E
                F
        C
        D
A: Starting...
C: Starting...
C: Sleeping...
D: Starting...
D: Sleeping...
B: Starting...


E: Starting...
E: Sleeping...
F: Starting...
F: Sleeping...
A(22626)─┬─B(22628)─┬─E(22631)
         │          └─F(22632)
         ├─C(22629)
         └─D(22630)


C: Exiting...
My PID = 22626: Child PID = 22629 terminated normally, exit status = 8
D: Exiting...
My PID = 22626: Child PID = 22630 terminated normally, exit status = 8
F: Exiting...
My PID = 22628: Child PID = 22632 terminated normally, exit status = 8
E: Exiting...
My PID = 22628: Child PID = 22631 terminated normally, exit status = 8
B: Exiting...
My PID = 22626: Child PID = 22628 terminated normally, exit status = 10
A: Exiting...
My PID = 22625: Child PID = 22626 terminated normally, exit status = 10
oslab122@orion:~/ex2$
```

2η εκτέλεση:

```
oslab122@orion:~/ex2$ ./tree-example proc.tree
A
        B
                E
                F
        C
        D
A: Starting...
D: Starting...
D: Sleeping...

C: Starting...
C: Sleeping...

B: Starting...
F: Starting...
F: Sleeping...
E: Starting...
E: Sleeping...
A(22658)─┬─B(22660)─┬─E(22664)
         │          └─F(22665)
         ├─C(22661)
         └─D(22662)


D: Exiting...
My PID = 22658: Child PID = 22662 terminated normally, exit status = 8
C: Exiting...
My PID = 22658: Child PID = 22661 terminated normally, exit status = 8
F: Exiting...
My PID = 22660: Child PID = 22665 terminated normally, exit status = 8
E: Exiting...
My PID = 22660: Child PID = 22664 terminated normally, exit status = 8
B: Exiting...
My PID = 22658: Child PID = 22660 terminated normally, exit status = 10
A: Exiting...
My PID = 22657: Child PID = 22658 terminated normally, exit status = 10
oslab122@orion:~/ex2$
```

- Ερωτήσεις:

Παρατηρούμε πως τα μηνύματα εξόδου των διεργασιών δε φαίνεται να ακολουθούν συγκεκριμένη σειρά, εξαρτάται απο τον CPU scheduler του υπολογιστή. Τα μηνύματα εκκίνησης φαίνεται να εκτελούν BFS traversal.

1.3 Αποστολή και χειρισμός σημάτων

Πηγαίος κώδικας:

```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"




void fork_procs(struct tree_node *root) {
   /*
    * Start
    */
   int status,i;
   printf("PID = %ld, name %s, starting...\n",
          (long) getpid(), root->name);
   change_pname(root->name);


   if (root->nr_children == 0) {

       raise(SIGSTOP); /* Stop yourself */
       exit(2); /* 2 = For leaf nodes */
   } else {
       pid_t pid[root->nr_children];
       for (i = 0; i < root->nr_children; i++) {

           pid[i] = fork();
           if (pid < 0) {
               perror("main: fork");
               exit(1);
           }
           if (pid[i] == 0) {
               fork_procs(root->children + i);
```

```
            }
        }


        wait_for_ready_children(root->nr_children);/* Wait for ready children */
        raise(SIGSTOP); /* Stop yourself */



        for (i = 0; i <root->nr_children; i++) {
            printf("Waking the process:%s...\n", root->children[i].name);
            kill(pid[i], SIGCONT);
            pid[i] = wait(&status);

            explain_wait_status(pid[i],status);

        }









        /*
         * Suspend Self
         */


        /* ... */

        /*
         * Exit
         */
        exit(8); /* 8 = For non-leaf nodes */
    }
}

/*
* The initial process forks the root of the process tree,
* waits for the process tree to be completely created,
* then takes a photo of it using show_pstree().
*
* How to wait for the process tree to be ready?
* In ask2-{fork, tree}:
*      wait for a few seconds, hope for the best.
* In ask2-signals:
*      use wait_for_ready_children() to wait until
```

```c
*       the first process raises SIGSTOP.
*/

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

    if (argc < 2){
        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) {
        perror("main: fork");
        exit(1);
    }
    if (pid == 0) {
        /* Child */
        fork_procs(root);
        exit(1);
    }

    /*
     * Father
     */
    /* for ask2-signals */
    wait_for_ready_children(1);

    /* for ask2-{fork, tree} */
    /* sleep(SLEEP_TREE_SEC); */

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

    /* for ask2-signals */
    printf("Waking the process:%s...\n", root->name);
    kill(pid, SIGCONT);

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

    return 0;
}
```

Κάνουμε αρχικά `make ask2-signals` και στη συνέχεια εκτελούμε με όρισμα το proc.tree

```
./ask2-signals proc.tree
```

Έχουμε output:

```
oslab122@orion:~/ex2$ ./ask2-signals proc.tree
PID = 5460, name A, starting...
PID = 5463, name D, starting...
My PID = 5460: Child PID = 5463 has been stopped by a signal, signo = 19
PID = 5461, name B, starting...
PID = 5465, name F, starting...
My PID = 5461: Child PID = 5465 has been stopped by a signal, signo = 19
PID = 5464, name E, starting...
My PID = 5461: Child PID = 5464 has been stopped by a signal, signo = 19
My PID = 5460: Child PID = 5461 has been stopped by a signal, signo = 19
PID = 5462, name C, starting...
My PID = 5460: Child PID = 5462 has been stopped by a signal, signo = 19
My PID = 5459: Child PID = 5460 has been stopped by a signal, signo = 19


A(5460)─┬─B(5461)─┬─E(5464)
        │         └─F(5465)
        ├─C(5462)
        └─D(5463)


Waking the process:A...
Waking the process:B...
Waking the process:E...
My PID = 5461: Child PID = 5464 terminated normally, exit status = 2
Waking the process:F...
My PID = 5461: Child PID = 5465 terminated normally, exit status = 2
My PID = 5460: Child PID = 5461 terminated normally, exit status = 8
Waking the process:C...
My PID = 5460: Child PID = 5462 terminated normally, exit status = 2
Waking the process:D...
My PID = 5460: Child PID = 5463 terminated normally, exit status = 2
My PID = 5459: Child PID = 5460 terminated normally, exit status = 8
oslab122@orion:~/ex2$
```

- Ερωτήσεις:

1. Τα signals ειναι ενας τροπος για IPC (Interprocess Communication) και λειτουργούν αποδοτικότερα σε αυτό το τομέα από τη χρήση του syscall της sleep(). Τα signals λειτουργούν γρηγορότερα στο να συγχρονίσουν διεργασίες καθώς στη περίπτωση της sleep() οι διεργασίες θα έπρεπε να “παγώσουν” για να συγχρονιστούν με αποτέλεσμα να χάνονται πολλοί κύκλοι CPU.
   Επίσης τα signals επιτρέπουν παραπάνω έλεγχο στις διεργασίες οπως continue (SIGCONT) , stop (SIGSTOP), kill (SIGKILL) σε αντίθεση με τη sleep() που δίνει μόνο fixed time delay.

2. Μπορούμε να δούμε τον ορισμό της wait_for_ready_children() μέσα στο proc-common.c και βγάζουμε τα εξής συμπεράσματα:
   Ως συνάρτηση περιμένει όλα τα παιδιά να έχουν σήμα SIGSTOP, να έχουν

“παγώσει” δηλαδή, προτού συνεχίσει να εκτελεί το υπόλοιπο πρόγραμμα. Αν δε γίνει αυτό και δε περιμένουμε όλα τα παιδιά, τότε ένα αποτέλεσμα είναι να μη τυπωθεί ολόκληρο το δέντρο αφου δε θα έχουμε σταματημένο κάθε παιδί της ρίζας. Αποφεύγεται έτσι κάποιο φαινόμενο race condition, π.χ σήμα SIGCONT πριν απο SIGSTOP στο παιδι.

```
/*
 * Make sure all the children have raised SIGSTOP,
 * by using waitpid() with the WUNTRACED flag.
 *
 * This will NOT work if children use pause() to wait for SIGCONT.
 */
void
wait_for_ready_children(int cnt)
{
        int i;
        pid_t p;
        int status;

        for (i = 0; i < cnt; i++) {
                /* Wait for any child, also get status for stopped children */
                p = waitpid(-1, &status, WUNTRACED);
                explain_wait_status(p, status);
                if (!WIFSTOPPED(status)) {
                        fprintf(stderr, "Parent: Child with PID %ld has died un
n",
                                (long)p);
                        exit(1);
                }
        }
}
```

1.4 Παράλληλος υπολογισμός αριθμητικής έκφρασης

Πηγαίος κώδικας:

```c
/*
* pipe-example.c
*/

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <assert.h>

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

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

```c
#include "tree.h"

void fork_procs(struct tree_node *root, int desc) {
   /*
    * Start
    */
   pid_t pid;
   int status;
   int val[2];
   int res,i;

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


   if (root->nr_children == 0) {
       int name=atoi(root->name);

       if(write(desc, &name, sizeof(name))!=sizeof(name)){
           perror("child number: write to pipe");
           exit(1);
       } /* Write the value into the pipe */
       close(desc);


       sleep(3);
       /* WE USE SLEEP IN ORDER TO PRINT THE PROC TREE, THERE IS NO */
       /* NEED FOR SYNCHRONIZATION */
       exit(2); /* 2 = For leaf nodes */
   }


   else {

       int pfd_child[2];
       if (pipe(pfd_child) < 0) {
           perror("pipe");
           exit(1);
       }
       for (int i = 0; i < root->nr_children; i++) {
           pid = fork();
           if (pid < 0) {
               perror("main: fork");
               exit(1);
           }
           if (pid == 0) {
               close(pfd_child[0]);
               fork_procs(root->children + i, pfd_child[1]);

           }
```

```
        }

        close(pfd_child[1]);



        for(i=0; i < root->nr_children; i++) { //wait for all children values

            if (read(pfd_child[0], &val[i], sizeof(val[i])) != sizeof(val[i])) {

                perror("read from pipe");
                exit(1);
            }
            printf("Parent %s: received value %d from the pipe.\n",
root->name,val[i]);

        }
        close(pfd_child[0]);


        /* Close the read end of the pipe */

        printf("Parent: received value1 %d and value2 %d from the pipe. Will now
compute.\n", val[0],val[1]);

        if (strcmp(root->name, "+") == 0) {
            res=val[0]+val[1];
            printf("Parent: %d + %d = %d,value1+value2\n",val[0],val[1],res);
        }
        else{
            res=val[0]*val[1];
            printf("Parent: %d * %d = %d,value1*value2\n",val[0],val[1],res);
        }

        if((write(desc, &res, sizeof(res)))!=sizeof(res)){
            perror("parent: write to pipe");
            exit(1);
        } /* Write the value into the pipe */

        close(desc);


        for (i = 0; i < root->nr_children; i++) {
            pid = wait(&status);

            explain_wait_status(pid, status);
        }
```

```
        exit(8); /* 8 = For non-leaf nodes */
    }

}



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

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

    printf("Parent: Creating pipe...\n");
    if (pipe(pfd) < 0) {
        perror("pipe");
        exit(1);
    }

    printf("Parent: Creating child...\n");
    p = fork();
    if (p < 0) {
        /* fork failed */
        perror("fork");
        exit(1);
    }
    if (p == 0) {
         close(pfd[0]);
        /* In child process */
         fork_procs(root,pfd[1]);
        /*
         * Should never reach this point,
         * child() does not return
         */

    }

    show_pstree(p);



    /* Wait for the child to terminate */
    printf("Parent: Created child with PID = %ld, waiting for it to
terminate...\n",
        (long)p);
```

```c
  if((read(pfd[0], &res, sizeof(res) )!= sizeof(res))){
      perror("parent: read from pipe");
      exit(1);
  } /* Read the value from the pipe */
  close(pfd[0]);

  printf("Final result: %d .\n", res);


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

  printf("Parent: All done, exiting...\n");

  return 0;
}
```

Κάνουμε αρχικά `make pipe-example` και στη συνέχεια εκτελούμε με όρισμα το proc.tree `./pipe-example expr.tree`

- Για την έκφραση 10+4*(5+7):

```
oslab122@orion:~/ex2$ ./pipe-example expr.tree
Parent: Creating pipe...
Parent: Creating child...
PID = 30116, name +, starting...


PID = 30119, name *, starting...
PID = 30118, name 10, starting...
PID = 30121, name +, starting...
PID = 30124, name 7, starting...
PID = 30123, name 5, starting...
PID = 30122, name 4, starting...
+(30116)─┬─*(30119)─┬─+(30121)─┬─5(30123)
         │          │          └─7(30124)
         │          └─4(30122)
         └─10(30118)


Parent: Created child with PID = 30116, waiting for it to terminate...
My PID = 30116: Child PID = 30118 terminated normally, exit status = 2
My PID = 30121: Child PID = 30123 terminated normally, exit status = 2
My PID = 30121: Child PID = 30124 terminated normally, exit status = 2
Parent +: received value 7 from the pipe.
Parent +: received value 5 from the pipe.
Parent: received value1 7 and value2 5 from the pipe. Will now compute.
Parent: 7 + 5 = 12,value1+value2
My PID = 30119: Child PID = 30121 terminated normally, exit status = 8
My PID = 30119: Child PID = 30122 terminated normally, exit status = 2
Parent *: received value 4 from the pipe.
Parent *: received value 12 from the pipe.
Parent: received value1 4 and value2 12 from the pipe. Will now compute.
Parent: 4 * 12 = 48,value1*value2
My PID = 30116: Child PID = 30119 terminated normally, exit status = 8
Parent +: received value 10 from the pipe.
Parent +: received value 48 from the pipe.
Parent: received value1 10 and value2 48 from the pipe. Will now compute.
Parent: 10 + 48 = 58,value1+value2
Final result: 58 .
My PID = 30115: Child PID = 30116 terminated normally, exit status = 8
Parent: All done, exiting...
```

Για την υλοποίηση αυτής της άσκησης δεν απαιτείται κάποιος συγχρονισμός των διεργασιών με sleep() ή σήματα και αυτό διότι όταν καλείται η syscall read() προκειμένου να διαβαστεί από τα παιδιά η αριθμητική τους τιμή η ίδια κάνει wait μέχρι να είναι διαθέσιμη κάποια τιμή. Χρησιμοποιούμε sleep() προκειμένου να τυπώσουμε το δέντρο διεργασιών, δεν είναι δηλαδή απαραίτητο για τον υπολογισμό.

- Ερωτήσεις:

1. Το pipe είναι one way τρόπος επικοινωνίας στο kernel space και είναι δομής FIFO. Η κάθε διεργασία μη-φύλλο απαιτεί μία σωλήνωση για όλα τα παιδιά της, σε αυτή γράφουν τα παιδιά τις αριθμητικές τους τιμές, απαιτείται ακόμα μία σωλήνωση ωστέ λειτουργεί ως συλλέκτης αποτελέσματος και μετακινεί προς τη ρίζα το αποτέλεσμα της πράξης του τελεστή. Η κοινή σωλήνωση για τα παιδιά είναι επιτρεπτή καθώς έχουμε commutative πράξεις.

2. Το πλεονέκτημα σε ένα σύστημα παράλληλων επεξεργαστών είναι η μείωση του χρόνου εκτέλεσης, ειδικότερα αν η έκφραση είναι μεγάλη, εκεί γίνεται εμφανέστερο το πλεονέκτημα, παράδειγμα τέτοιων συστημάτων είναι οι GPU οι οποίες παραλληλοποιούν κομμάτια κώδικα και επιταχύνουν σημαντικά τη χρονική πολυπλοκότητα.

Προαιρετικές Ερωτήσεις

1. Οι παράγοντες που μπορεί να επηρεάσουν την απόφαση μας για το αν η παραλληλοποίηση μιας αριθμητικής έκφρασης είναι πάντα προτιμότερη απο την εκτέλεση της ως μία διεργασία είναι:

- Πολυπλοκότητα έκφρασης: Επειδή η παραλληλοποίηση απαιτεί διαδιεργασιακή επικοινωνία και επομένως έχει overhead σε πολύ πολύπλοκες εκφράσεις μπορεί η επιτάχυνση της παραλληλοποίησης να είναι αμελητέα ή ακόμα και να επιβραδύνει τον υπολογισμό της έκφρασης.

- Contention: Οταν πολλοί επεξεργαστές έχουν πρόσβαση στο ίδιο κομμάτι μνήμης μπορεί να καταλήξουμε να έχουμε contention για πρόσβαση σε δεδομένα και εγγραφή στη μνήμη που μπορεί σε ακραίες περιπτώσεις να οδηγήσει σε επιβράδυνση λόγω πολλαπλών wait που θα κάνουν οι ξεχωριστές διεργασίες.