Σχολή Ηλεκτρολόγων Μηχανικών

&

Μηχανικών Υπολογιστών

Εργαστήριο Λειτουργικών Συστημάτων

2η Εργαστηριακή Άσκηση

Καραμπάση Αικατερίνη, Α.Μ. : 03112517

Μανδηλαράς Νικηφόρος, Α.Μ: 03112012

Ομάδα : **D12**

Έβδομο Εξάμηνο

Παραδοτέα: 3/12 /2015

1.1 Δημιουργία δεδομένου δέντρου διεργασιών

Ο κώδικας μας παράγει το δεδομένο δέντρο που ζητείται από την εκφώνηση κάνοντας διαδοχική χρήση συναρτήσεων fork για να παράγει και να αρχικοποιήσει νέους κόμβους του δέντρου. Οι διεργασίες θέλουμε να δημιουργούνται και να παραμένουν ενεργές για συγκεκριμένο χρονικό διάστημα. Κάθε γονιός περιμένει πρώτα τα παιδιά του να τερματιστούν, κάνοντας wait, πριν επιστρέψει και ο ίδιος. Μόλις όλοι οι κόμβοι έχουν επιστρέψει η αρχική μας διαδικασία τυπώνει το δέντρο που δημιουργήσαμε και στη συνέχεια τερματίζει.

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

/*
 * Create this process tree:
 * A-+-B---D
 *   `-C
 */

void fork_procs(void)
{
    pid_t pid;
       int status;

       printf("A: Starting...\n");
       change_pname("A");
       pid = fork();              //Create B
       if (pid < 0)
       {
              perror("A fork B");
              exit(1);
       }
       else if (pid == 0)
       {
              printf("A: Waiting...\n");//A has 2 children
              printf("B: Starting...\n");
              change_pname("B");

              pid = fork();       //Create D
              if (pid < 0)
              {
                     perror("B fork D");
                     exit(1);
              }
              else if (pid == 0)
              {
                     printf("B: Waiting...\n"); //B is waiting for show_pstree
                     printf("D: Starting...\n");
                     change_pname("D");
                     printf("D: Sleeping...\n"); //sleep for show_pstree
                     sleep(SLEEP_PROC_SEC);
                     printf("D: Exiting...\n");
                     exit(13);
              }
              pid = wait(&status);        //B has a child, so one wait
```

```
                explain_wait_status(pid,status);
                printf("B: Exiting...\n");
                exit(19);
        }

        pid = fork();              //Create C
        if (pid < 0)
        {
                perror("A fork C");
                exit(1);
        }
        else if (pid == 0)
        {
                printf("C: Starting...\n");
                change_pname("C");
                printf("C: Sleeping...\n");             //sleep for show_pstree
                sleep(SLEEP_PROC_SEC);
                printf("C: Exiting...\n");
                exit(17);
        }
        pid = wait(&status);              //A has 2 wait one for
        explain_wait_status(pid,status);        //each children
        pid = wait(&status);
        explain_wait_status(pid,status);
        printf("A: Exiting...\n");
        exit(16);
}



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

    pid = fork();   //Create process A
    if (pid < 0) {  //the root of our processes
        perror("main: fork");
        exit(1);
    }
    if (pid == 0) { //After process is ready root-A
        fork_procs();      //We are going to create the tree
        exit(1);
    }
       printf("Start!\n");
       sleep(SLEEP_TREE_SEC);     //Father is waiting until
       show_pstree(pid);
       pid = wait(&status);       //Wait until it's complited
       explain_wait_status(pid, status);
       printf("Exit!\n");
       return 0;
}
```

Ενδεικτικά η έξοδος που παίρνουμε από την εκτέλεση της πρώτης άσκησης είναι η εξής, αφού ξέρουμε πόσα και ποια παιδιά θέλουμε να δημιουργήσουμε:

Start!
A: Starting...
A: Waiting...
B: Starting...
C: Starting...
C: Sleeping...
B: Waiting...
D: Starting...
D: Sleeping...

```
A(2891)─┬─B(2892)──D(2894)
        └─C(2893)
```

```
C: Exiting...
My PID = 2891: Child PID = 2893 terminated normally, exit status = 17
D: Exiting...
My PID = 2892: Child PID = 2894 terminated normally, exit status = 13
B: Exiting...
My PID = 2891: Child PID = 2892 terminated normally, exit status = 19
A: Exiting...
My PID = 2890: Child PID = 2891 terminated normally, exit status = 16
Exit!
```

Ερωτήσεις:

1. Αν τερματιστεί η διεργασία A με σήμα Kill τότε τα παιδιά της B , C θα κληρονομηθούν από την init που εκτελεί πάντα wait (τα παιδιά των B και C θα παραμείνουν παιδιά τους). Κατά την εμφάνιση του δένδρου δε θα εμφανιστεί τίποτα εκτός αν σκοτώσουμε την A προτού η γονική της διαδικασία κάνει wait οπότε η A θα γίνει zombie και θα εμφανιστεί.

2. Η συνάρτηση get_pid() επιστρέφει το id της διαδικασίας της οποίας βρισκόμαστε τη δεδομένη στιγμή , συνεπώς επιστρέφει την id του γονέα του A και εμφανίζεται αυτή η διαδικασία ως ρίζα του δένδρου. Τα παιδιά της είναι η A με το υποδέντρο που δημιουργήσαμε προηγουμένως καθώς και η διεργασία sh που έχει ένα επιπλέον παιδί την pstree. Αυτές οι διεργασίες δημιουργούνται από την show_pstree().

3. Τίθενται περιορισμοί καθώς αλλιώς θα μπορούσε ένας χρήστης να δημιουργήσει μεγάλο αριθμό διαδικασιών και λόγο του ότι ο χρόνος εξυπηρέτησης της κάθε διεργασίας ισοκατανέμεται οι χρήστες με περιορισμένες διεργασίες θα αδικούνταν. Επίσης ο περιορισμός τίθεται και για λόγους ασφαλείας ώστε κάποιος να μην μπορεί να δημιουργήσει μεγάλο αριθμό διαδικασιών για κακόβουλο σκοπό.

1.2 Δημιουργία αυθαίρετου δένδρου διεργασιών

Το πρόγραμμά μας δέχεται ως είσοδο αρχείο που περιέχει το ζητούμενο δέντρο και τυπώνει στην οθόνη το κατάλληλο δέντρο μαζί με τα αντίστοιχα μηνύματα.

Κώδικας:

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

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

#define SECS_PROC 4
#define SECS_MAIN 2

pid_t fork_procedure(struct tree_node *t)
{
       pid_t pid;
```

```
	int status,i;

	pid = fork();//create process
	if (pid<0)
	{
		perror("fork");
		exit(1);
	}
	else if (pid == 0)
	{
		change_pname(t->name);
		printf("Process %s created\n", t->name);
		if (t->nr_children == 0)
		{	//if the node has no children, the process is sleeping
			printf("%s is sleeping...\n", t->name);
			sleep(SECS_PROC);
			printf("%s is exiting...\n", t->name);
			exit(0);
		}
		else
		{
			for (i=0; i< t->nr_children; i++)
			{
				fork_procedure(t->children+i); //recursively
			}
			printf("%s is waiting...\n", t->name);
			for (i=0; i< t->nr_children; i++)
			{	//for every node/process
				pid = wait(&status);//wait for all of its children
				explain_wait_status(pid, status);
			}
			printf("%s is exiting...\n", t->name);
			exit(0);
		}
	}
	return pid;
}


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

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

	root = get_tree_from_file(argv[1]); //read file

	if (!root)
		printf("The input file is empty\n");
	else
	{
	//	print_tree(root);
		pid = fork_procedure(root);
		sleep(SECS_MAIN);   //sleep for the creation of the tree
		show_pstree(pid);
		pid = wait(&status);       //wait for the root of the tree
		explain_wait_status(pid, status);
		printf("Exit!");
	}
	return 0;

}
```

Σε αυτό το ερώτημα δημιουργούμε αυθαίρετο δέντρο το οποίο μας δίνεται στο έγγραφο proc.tree οπότε γράφοντας την εντολή ./twotwo proc.tree έχουμε το εξής αποτέλεσμα:

```
Process A created
Process B created
Process C created
C is sleeping...
A is waiting...
Process D created
D is sleeping...
Process E created
E is sleeping...
B is waiting...
Process F created
F is sleeping...
```

```
A(2950)─┬─B(2951)─┬─E(2954)
        │         └─F(2955)
        ├─C(2952)
        └─D(2953)
```

```
C is exiting...
D is exiting...
E is exiting...
F is exiting...
My PID = 2950: Child PID = 2952 terminated normally, exit status = 0
My PID = 2951: Child PID = 2954 terminated normally, exit status = 0
My PID = 2951: Child PID = 2955 terminated normally, exit status = 0
My PID = 2950: Child PID = 2953 terminated normally, exit status = 0
B is exiting...
My PID = 2950: Child PID = 2951 terminated normally, exit status = 0
A is exiting...
My PID = 2949: Child PID = 2950 terminated normally, exit status = 0
Exit!
```

Ερώτηση:

1. Τα μηνύματα κατά την εκκίνηση εμφανίζονται κατά BFS τρόπο, χωρίς όμως η σειρά να είναι απόλυτη ως προς το ποιο παιδί από κάποιο επίπεδο θα εμφανιστεί πριν από κάποιο άλλο, λόγο του γεγονότος ότι κατά την εκτέλεση της fork δε γνωρίζουμε σε ποιο σημείο του προγράμματος θα μεταβεί ο έλεγχος (στον γονιό ή στο παιδί). Κατά τον τερματισμό όμως των διαδικασιών τα καταληκτικά μηνύματα εμφανίζονται με DFS σειρά το οποίο εξασφαλίζεται από το γεγονός ότι κάθε γονιός περιμένει αναδρομικά να τελειώσει ένα ένα όλα του τα παιδιά.

1.3 Αποστολή και χειρισμός σημάτων

Το πρόγραμμά μας εκτελεί την ίδια διεργασία με πριν με την διαφορά ότι σε αυτή την υλοποίηση για τον συγχρονισμό των διεργασιών χρησιμοποιεί σήματα που αποστέλλονται μεταξύ των διεργασιών αντί για των συναρτήσεων wait.

Κώδικας:

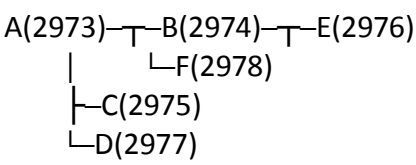
```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>

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

void fork_procs(struct tree_node *root)
{

	pid_t p[root->nr_children],pchild;
	int status,i;

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

	//does not enter on leaves
	for (i=0; i < root->nr_children; i++)
		{
			p[i] = fork();
			if (p[i]<0){
				perror("fork");
				exit(1);
			}
			else if (p[i] == 0){
				//recursively create children
				fork_procs(root->children+i);
				//no recursion = error
				exit(1);
			}
			else{
				printf("Parent, PID = %ld: Created child with PID = %ld,
waiting for it to terminate...\n", (long)getpid(), (long)p[i]);
			}
		}

		//all children must be STOPPED
		wait_for_ready_children(root->nr_children);
		printf("PID = %ld, name = %s has stopped\n", (long)getpid(), root->name);

		//when all children are STOPPED, STOP self
		raise(SIGSTOP);
		//when CONTINUED
		printf("PID = %ld, name = %s is awake\n", (long)getpid(), root->name);

		//SIGCONT all children
		for (i=0; i < root->nr_children; i++)
			{
				kill(p[i],SIGCONT);
				pchild = wait(&status);
				//wait for them to exit
				explain_wait_status(pchild,status);
			}

		//all processes exit normally in the end
		printf("%s: Exiting...\n",root->name);
		exit(0);
}

/*
 * 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(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]); //READ INPUT

       /* 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);

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

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

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

       return 0;
}
```

Στη συγκεκριμένη άσκηση εκτελούμε το πρόγραμμά μας πληκτρολογώντας την εντολή ./ask2-signals proc.tree και μας εμφανίζει:

PID = 2973, name A, starting...
Parent, PID = 2973: Created child with PID = 2974, waiting for it to terminate...
PID = 2974, name B, starting...
Parent, PID = 2973: Created child with PID = 2975, waiting for it to terminate...
Parent, PID = 2974: Created child with PID = 2976, waiting for it to terminate...
Parent, PID = 2973: Created child with PID = 2977, waiting for it to terminate...
Parent, PID = 2974: Created child with PID = 2978, waiting for it to terminate...

PID = 2977, name D, starting...
PID = 2975, name C, starting...
PID = 2975, name = C has stopped
PID = 2977, name = D has stopped
PID = 2978, name F, starting...
PID = 2978, name = F has stopped
PID = 2976, name E, starting...
PID = 2976, name = E has stopped
My PID = 2973: Child PID = 2975 has been stopped by a signal, signo = 19
My PID = 2974: Child PID = 2978 has been stopped by a signal, signo = 19
My PID = 2973: Child PID = 2977 has been stopped by a signal, signo = 19
My PID = 2974: Child PID = 2976 has been stopped by a signal, signo = 19
PID = 2974, name = B has stopped
My PID = 2973: Child PID = 2974 has been stopped by a signal, signo = 19
PID = 2973, name = A has stopped
My PID = 2972: Child PID = 2973 has been stopped by a signal, signo = 19



PID = 2973, name = A is awake
PID = 2974, name = B is awake
PID = 2976, name = E is awake
E: Exiting...
My PID = 2974: Child PID = 2976 terminated normally, exit status = 0
PID = 2978, name = F is awake
F: Exiting...
My PID = 2974: Child PID = 2978 terminated normally, exit status = 0
B: Exiting...
My PID = 2973: Child PID = 2974 terminated normally, exit status = 0
PID = 2975, name = C is awake
C: Exiting...
My PID = 2973: Child PID = 2975 terminated normally, exit status = 0
PID = 2977, name = D is awake
D: Exiting...
My PID = 2973: Child PID = 2977 terminated normally, exit status = 0
A: Exiting...
My PID = 2972: Child PID = 2973 terminated normally, exit status = 0

Ερωτήσεις:

1. Ουσιαστικά αυτό που έκανε η συνάρτηση sleep ήταν να θέτει σε αναστολή τη λειτουργία μιας διεργασίας για ένα συγκεκριμένο και προκαθορισμένο χρονικό διάστημα προκειμένου να δημιουργηθούν άλλες διεργασίες και να εκτελέσουν συγκεκριμένες ενέργειες . Ενώ με τα σήματα μεταξύ των διαδικασιών μπορούμε αμέσως μόλις έχει ολοκληρωθεί η απαιτούμενη ενέργεια, να ξυπνήσουμε τότε ακριβώς τη διαδικασία που έχει τεθεί σε αναστολή.

2. Αυτό που κάνει η wait_for_ready_children() είναι να δέχεται ως όρισμα το πλήθος των παιδιών μιας διαδικασίας και να περιμένει καθένα από αυτά να αναστείλει τη λειτουργία του. Αν είχαμε αγνοήσει την εκτέλεση αυτής της συνάρτησης τότε δεν είμαστε σίγουροι για το αν όλα τα παιδιά μιας διεργασίας έχουν σταματήσει οπότε αν ο γονέας στείλει σήμα SIGCONT προς κάποιο παιδί που δεν έχει εκτελέσει SIGSTOP τότε το σήμα αυτό θα αγνοηθεί.

1.4 Παράλληλος υπολογισμός αριθμητικής έκφρασης

Στο πρόγραμμα αυτό επεκτείνεται η υλοποίηση 1.2 ώστε στο αρχείο εισόδου να περιέχεται μια αριθμητική έκφραση. Κατά την εκτέλεση του προγράμματος εμφανίζονται τα κατάλληλα μηνύματα ανάλογα με τον αν ένας κόμβος είναι τελεστής ή νούμερο και τέλος αποτιμάται το αποτέλεσμα της έκφρασης.

Κώδικας:

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

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

#define SLEEP_TREE_SEC  2
#define SLEEP_PROC_SEC  4

void fork_procs(struct tree_node *root, int fd)
{
	int pfd1[2], pfd2[2], status, i, num, num1, num2;
	pid_t p;

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

	//if it is a parent...create children process tree...leaves do not enter this
	if (root->nr_children == 2)
	{
		//pipe of first child
		if (pipe(pfd1) < 0) {
			perror("pipe");
			exit(1);
		}
		//create first child
		p = fork();
		if (p < 0) {
			perror("fork");
			exit(1);
		}
		if (p == 0) {
			/* First Child */
			//child 1 can not read
			close(pfd1[0]);
			//recursively create children and give them the "write" end of pipe

			fork_procs(root->children, pfd1[1]);
			exit(1);
		}
		//father can not write
		close(pfd1[1]);
```

```
		//pipe of second child
		if (pipe(pfd2) < 0) {
			perror("pipe");
			exit(1);
		}
		//create second child
		p = fork();
		if (p < 0) {
			perror("fork");
			exit(1);
		}
		if (p == 0) {
			/* Second Child */
			//child 2 does not read
			close(pfd2[0]);
			//recursively create children and give them the "write" end of pipe

			fork_procs(root->children+1, pfd2[1]);
			exit(1);
		}
		//father does not write
		close(pfd2[1]);
	}
	//leaf = name is an integer
	if (root->nr_children == 0)
	{
		//wait for show_pstree like ask2-tree
		printf("%s is sleeping...\n", root->name);
		sleep(SLEEP_PROC_SEC);
		//number calculation
		num = atoi(root->name);    // oxi atoi
		printf("Leaf: %s, PID: %ld, gave number %d...\n",root->name,
(long)getpid(), num);
		//give number to parent through pipe
		if (write(fd, &num, sizeof(num)) != sizeof(num)) {
			perror("leaf: write to pipe");
			exit(1);
		}
	}
	//parent = name is mathematical symbol
	else if (root->nr_children == 2)
	{
		//read num1
		if (read(pfd1[0], &num1, sizeof(num1)) != sizeof(num1)) {
			perror("parent: read from pipe");
			exit(1);
		}
		printf("Parent: %s, PID: %ld, received number %d...\n",root->name,
(long)getpid(), num1);
		//read num2
		if (read(pfd2[0], &num2, sizeof(num2)) != sizeof(num2)) {
			perror("parent: read from pipe");
			exit(1);
		}
		printf("Parent: %s, PID: %ld, received number %d...\n",root->name,
(long)getpid(), num2);
		//choose mathematical symbol
		if (!strcmp(root->name,"+"))
		{
			printf("Parent: %s, PID: %ld, I am an adder...\n",root->name,
(long)getpid());
			//compute
			num = num1 + num2;
		}
		//choose mathematical symbol
		else if (!strcmp(root->name,"*"))
		{
			printf("Parent: %s, PID: %ld, I am a multiplier...\n",root->name,
(long)getpid());
			//compute
```

```
                        num = num1 * num2;
                }
                printf("Parent: %s, PID: %ld, I calculated %d...\n",root->name,
(long)getpid(), num);
                //give result to parent through pipe
                if (write(fd, &num, sizeof(num)) != sizeof(num)) {
                        perror("Parent: write to pipe");
                        exit(1);
                }
                //wait for children to exit
                for (i = 0; i < root->nr_children; i++){
                        p = wait(&status);
                        explain_wait_status(p,status);
                }
        }

        /*
         * Exit
         */
        //all processes exit normally in the end
        printf("Process %s, PID: %ld, Exiting...\n",root->name, (long)getpid());
        exit(0);
}

/*
 * 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(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 */
        int pfd[2];
        //pipe of main process
        if (pipe(pfd) < 0) {
                perror("pipe");
                exit(1);
        }

        pid = fork();
        if (pid < 0) {
                perror("main: fork");
                exit(1);
        }
        if (pid == 0) {
                /* Child */
                //child does not read
                close(pfd[0]);
                //recursively create children and give them the "write" end of pipe

                fork_procs(root,pfd[1]);
```

```
                exit(1);
        }

        /*
         * Father
         */
        //father does not write
        close(pfd[1]);
        //wait for tree creation
        sleep(SLEEP_TREE_SEC);

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

        int value;
        //main process receives final value
        if (read(pfd[0], &value, sizeof(value)) != sizeof(value)) {
                perror("main: read from pipe");
                exit(1);
        }
        //print result
        printf("Final Result: %d...\n",value);
        /* Wait for the root of the process tree to terminate */
        pid = wait(&status);
        explain_wait_status(pid, status);

        return 0;
}
```

Τέλος, σε αυτό το ερώτημα δημιουργήσαμε ένα αρχείο με όνομα input το οποίο και τρέξαμε με την εντολή ./ask2_4 input και μας εμφανίζει το εξής:

PID = 2996, name *, starting...
PID = 2997, name +, starting...
PID = 2998, name 10, starting...
10 is sleeping...
PID = 2999, name 5, starting...
5 is sleeping...
PID = 3000, name 6, starting...
6 is sleeping...

```
*(2996)─┬─+(2997)─┬─5(2999)
        |         └─6(3000)
        └─10(2998)
```

Leaf: 10, PID: 2998, gave number 10...
Leaf: 5, PID: 2999, gave number 5...
Process 10, PID: 2998, Exiting...
Process 5, PID: 2999, Exiting...
Leaf: 6, PID: 3000, gave number 6...
Parent: +, PID: 2997, received number 5...
Process 6, PID: 3000, Exiting...
Parent: +, PID: 2997, received number 6...
Parent: +, PID: 2997, I am an adder...
Parent: +, PID: 2997, I calculated 11...
Parent: *, PID: 2996, received number 11...

Parent: *, PID: 2996, received number 10...
Parent: *, PID: 2996, I am a multiplier...
Parent: *, PID: 2996, I calculated 110...
Final Result: 110...
My PID = 2996: Child PID = 2998 terminated normally, exit status = 0
My PID = 2997: Child PID = 2999 terminated normally, exit status = 0
My PID = 2997: Child PID = 3000 terminated normally, exit status = 0
Process +, PID: 2997, Exiting...
My PID = 2996: Child PID = 2997 terminated normally, exit status = 0
Process *, PID: 2996, Exiting...
My PID = 2995: Child PID = 2996 terminated normally, exit status = 0

Ερωτήσεις:

1. Στην υλοποίηση μας χρησιμοποιούμε μία σωλήνωση για κάθε διεργασία. Είναι γεγονός πως θα μπορούσαμε να χρησιμοποιούμε μία σωλήνωση αποδίδοντας την πρώτα στο ένα παιδί και έπειτα στο άλλο επειδή οι πράξεις που πραγματοποιούμε είναι αντιμεταθετικές. Στην περίπτωση αφαίρεσης ή διαίρεσης προκειμένου να υλοποιηθούν με μία μόνο σωλήνωση θα έπρεπε τα παιδιά να γράφουν στο pipe πέρα του αριθμού τους και το pid τους. Ο γονιός είναι σε θέση να γνωρίζει τα pid των παιδιών του και τη σειρά με την οποία τα δημιούργησε οπότε και θα εκτελέσει την πράξη σωστά.

2. Στην περίπτωση πολλών επεξεργαστών μπορούν να γίνουν πολλές διαδικασίες παράλληλα και αυτό έχει συνέπεια η έκφραση να αποτιμάται σε λιγότερο χρόνο.

Προαιρετικές ερωτήσεις:

1) Ο βασικός παράγοντας που επηρεάζει την ταχύτερη εκτέλεση είναι πως όταν διαθέτουμε έναν επεξεργαστή τότε εκτελείται μία διεργασία κάθε φορά, ενώ στην περίπτωση των πολλών επεξεργαστών έχουμε παράλληλη εκτέλεση των διεργασιών κάτι που εκμεταλλευόμαστε ώστε να έχουμε πιο άμεσα τα αποτελέσματα.
2) Το πλεονέκτημα που μπορεί να μας προσφέρει η υβριδική υλοποίηση είναι πως όσες διεργασίες θα απαιτήσουν τον πιο πολύ χρόνο προκειμένου να εκτελεστούν θα το κάνουν αξιοποιώντας την επιλογή να εκτελεστούν διαχωρισμένες. Ωστόσο, όταν έρθει η ώρα να αξιοποιηθούν τα αποτελέσματα των επιμέρους διεργασιών στο τελικό αποτέλεσμα τότε αυτό θα γίνει πιο εύκολα σε μία διεργασία έχοντας τα αποτελέσματα από προηγουμένως. Η παράμετρος μ, επομένως, θα πρέπει να έχει τέτοια τιμή που να καταφέρνει να μας αποφέρει κέρδος χρόνου σε σχέση με την εκτέλεση χωρίς την υβριδική υλοποίηση. Δηλαδή, να γίνει με τέτοιο τρόπο ώστε να είναι συμφέρουσα σε σχέση με το να μην χρησιμοποιούσαμε το συνδυασμό των παράλληλων και σειριακών διεργασιών. Ο παράγοντας που θα επηρεάσει την τιμή του μ είναι κατά πόσο μας συμφέρει να κρατήσουμε την παράλληλη εκτέλεση των διεργασιών και σε ποιο σημείο μας συμφέρει να εντάξουμε στη διαδικασία τη σειριακή εκτέλεση.