Λειτουργικά Συστήματα Υπολογιστών

Αναφορά - 2η Εργαστηριακή Άσκηση

Ομάδα: oslab04

Κατσιάνης Κωνσταντίνος - 03120183

Κουρής Γεώργιος - 03120116

6ο Εξάμηνο - Απρίλιος 2023

Περιεχόμενα Αναφοράς:

2 Αναφορά άσκησης

1.1 Σύνδεση με αρχείο αντικειμένων

Α) Ο πηγαίος κώδικας (source code) της άσκησης

```
#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

void fork_procs(void)
{
	pid_t Bpid, Cpid, Dpid, kill_child;
	int waitstatus;
	change_pname("A");
	printf("A: Creating...\n");

	Bpid=fork();

	if (Bpid < 0) {
		perror("fork-error");
		exit(1);
	}
	else if (Bpid == 0) {
		change_pname("B");
```

```
        printf("B: Creating...\n");
        Dpid=fork();
        if (Dpid < 0) {
            perror("fork-error");
            exit(1);
        }
        else if (Dpid == 0) {
            change_pname("D");
            printf("D: Creating...\n");
            printf("D: Sleeping...\n");
            sleep(SLEEP_PROC_SEC);
            printf("D: Exiting...\n");
            exit(13); //return code for D
        }
        else  {
            printf("B: Waiting...\n");
            kill_child = wait(&waitstatus);
            if(kill_child < 0) {
                perror("wait-error");
                exit(1);
            }
            explain_wait_status(kill_child, waitstatus);
            printf("B: Exiting...\n");
            exit(19); //return code for B
        }
    }
    else  {
        Cpid=fork();
        if (Cpid < 0) {
            perror("fork-error");
            exit(1);
        }
```

```
        else if (Cpid == 0) {
            change_pname("C");
            printf("C: Creating...\n");
            printf("C: Sleeping\n");
            sleep(SLEEP_PROC_SEC);
            printf("C: Exiting\n");
            exit(17); //return code for C
        }
        else {
            printf("A: Waiting...\n");
            kill_child = wait(&waitstatus);
            if(kill_child < 0) {
                perror("wait-error");
                exit(1);
            }
            explain_wait_status(kill_child, waitstatus);

            kill_child = wait(&waitstatus);
            if (kill_child < 0) {
                perror("wait-error");
                exit(1);
            }
            explain_wait_status(kill_child, waitstatus);
        }
    }
    printf("A: Exiting...\n");
    exit(16);
}
int main(void)
{
    pid_t pid;
    int status;
```

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

	sleep(SLEEP_TREE_SEC);

	show_pstree(pid);

	pid = wait(&status);
	if (pid < 0) {
		perror("wait-error");
		exit(1);
	}
	explain_wait_status(pid, status);
	return 0;
}
```

Β) Η έξοδος εκτέλεσης του προγράμματος

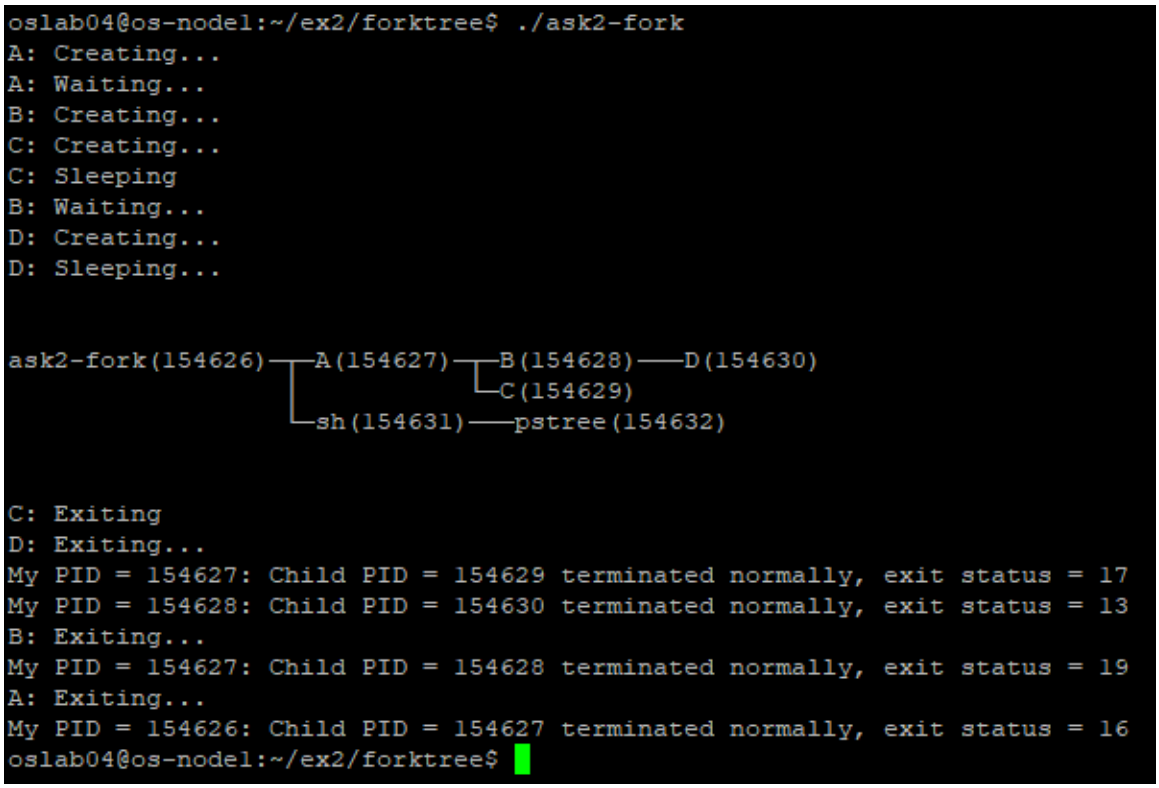
Για τη μεταγλώττιση χτησιμοποιήσαμε το Makefile που μας δίνεται από το αρχείο forktree χωρίς αλλαγές. Το αποτέλεσμα όταν τρέχουμε το ask2-fork είναι:

```
oslab04@os-node1:~/ex2/forktree$ ./ask2-fork
A: Creating...
A: Waiting...
B: Creating...
C: Creating...
C: Sleeping
B: Waiting...
D: Creating...
D: Sleeping...


A(154486)─┬─B(154487)──D(154489)
          └─C(154488)


C: Exiting
D: Exiting...
My PID = 154486: Child PID = 154488 terminated normally, exit status = 17
My PID = 154487: Child PID = 154489 terminated normally, exit status = 13
B: Exiting...
My PID = 154486: Child PID = 154487 terminated normally, exit status = 19
A: Exiting...
My PID = 154485: Child PID = 154486 terminated normally, exit status = 16
oslab04@os-node1:~/ex2/forktree$
```

Γ) Σύντομες απαντήσεις στις ερωτήσεις (1,2,3)

1) Αν τερματίσουμε πρόωρα τη διεργασία A, δίνοντας kill -KILL <pid>, όπου <pid> το Process ID της, τότε η εντολή -KILL στέλνει το signal SIGKILL στη διεργασία, που την τερματίζει άμεσα. Όλα τα child proccesses αυτής της parent proccess που πεθαίνει θα γίνουν ορφανά. Αυτά τα ορφανά θα τα κληρονομήσει η init, που κάνει συνεχώς wait.

2) Κάνοντας την αλλαγή στο πρόγραμμα παίρνουμε το ακόλουθο αποτέλεσμα:



Χρησιμοποιώντας το getpid() αντί για το pid μας εμφανίζεται όλο το δέντρο διεργασιών, με ρίζα την διεργασία ask2-fork. Στο δέντρο περιλαμβάνονται το δέντρο που εμφανιζόταν πριν αλλαγή της εντολής και οι διεργασία sh, η οποία έχει ως παιδί το pstree. Αυτές καλούνται από την show_pstree.

3) Σε ένα σύστημα με πολλούς χρήστες είναι σημαντικό ο διαχειριστής να θέτει όρια στον αριθμό των διεργασιών που μπορεί να δημιουργήσει ένας χρήστης. Με αυτό τον τρόπο δεν μπορεί ένα χρήστης να επιβαρύνει το σύστημα για τους υπόλοιπους χρήστες εκτελώντας ένα μεγάλο αριθμό διεργασιών. Θα μπορούσε για παράδειγμα να εκτελέσει έναν άπειρο αριθμό διεργασιών μέσω μιας απλής fork loop, κάνοντας αδύνατη την χρήση του συστήματος για τους υπόλοιπους χρήστες. Αυτό φυσικά θέλουμε να το αποφύγουμε.

1.2 Δημιουργία αυθαίρετου δέντρου διεργασιών

Α) Ο πηγαίος κώδικας (source code) της άσκησης

```
#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 SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3

void fork_procs(struct tree_node *node) {
	int wait_status;
	pid_t pid_child;
	change_pname(node->name);
	printf("%s: Created\n", node->name);
	if (node->nr_children == 0) {
		printf("%s: Sleeping...\n", node->name);
		sleep(SLEEP_PROC_SEC);
		printf("%s: Exiting...\n", node->name);
		exit(17); //value is arbitrary
	}
	int i;
	for (i = 0; i < node->nr_children; i++) {
		pid_child = fork();
		if (pid_child < 0) {
			perror("fork-error");
			exit(1);
		}
		if(pid_child == 0) {
			fork_procs(node->children + i);
```

```
            }
        }
        printf("%s: Waiting...\n", node->name);
        for (i=0; i < node->nr_children; i++) {
            pid_child = wait(&wait_status);
            if (pid_child < 0) {
                perror("wait-error");
                exit(1);
            }
            explain_wait_status(pid_child, wait_status);
        }
        printf("%s: Exiting...\n", node->name);
        exit(21); //value is arbitrary
}

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

        if (argc != 2) {
            printf("Usage: ./<file> <tree_file>\n");
            exit(1);
        }

        root = get_tree_from_file(argv[1]);
        printf("Print tree:\n");
        print_tree(root);
        printf("\n");

        pid = fork();
```

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

        sleep(SLEEP_TREE_SEC);
        show_pstree(pid);
        pid = wait(&status);
        explain_wait_status(pid, status);
        return 0;
}
```

Β) Η έξοδος εκτέλεσης του προγράμματος

Η έξοδος που παίρνουμε για το αρχείο proc.tree είναι:

```
oslab04@orion:~/ex2/forktree2$ ./ask2-tree-procs proc.tree
Print tree:
A
        B
                E
                F
        C
        D

A: Created
A: Waiting...
D: Created
D: Sleeping...
B: Created
B: Waiting...
C: Created
C: Sleeping...
F: Created
F: Sleeping...
E: Created
E: Sleeping...


A(20429)─┬─B(20430)─┬─E(20433)
         │          └─F(20434)
         ├─C(20431)
         └─D(20432)


D: Exiting...
C: Exiting...
F: Exiting...
E: Exiting...
B: Exiting...
A: Exiting...
My PID = 20428: Child PID = 20429 terminated normally, exit status = 21
oslab04@orion:~/ex2/forktree2$
```

Τροποποιήσαμε λίγο το Makefile, ώστε να επιτύχουμε linking και με το tree.h αρχείο επιπλέον:

```
.PHONY: all clean

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

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

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

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

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


ask2-signals: ask2-signals.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-{tree-procs,tree,signals,pipes}
~
~
```

Γ) Σύντομη απάντηση στην ερώτηση (1)

1) Γενικά, δεν είναι ντετερμινιστική η σειρά με την οποία εμφανίζονται τα μηνύματα “created”, “waiting”, “sleeping” και τα μηνύματα εξόδου. Παρόλα αυτά, παρατηρούμε ότι στην προκειμένη περίπτωση τα μηνύματα “created” εμφανίζονται με BFS τρόπο, και τα μηνύματα εξόδου αντίστροφα. Δηλαδή, εμφανίζονται με αυτόν τον τρόπο λόγω του τρόπου με τον οποίο διαχειρίζεται ο scheduler τις διεργασίες του.

1.3 Αποστολή και χειρισμός σημάτων

Α) Ο πηγαίος κώδικας (source code) της άσκησης

```
#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)
{
	int wait_status;
	pid_t pid_child[root->nr_children];
	printf("PID = %ld, name %s, starting...\n",(long)getpid(), root->name);
	change_pname(root->name);

	if (root->nr_children == 0) {
		raise(SIGSTOP);
		printf("PID = %ld, name = %s is awake\n",(long)getpid(), root->name);
		exit(15); //arbitrary value for leaf procs
	}

	else {
		int i;
		for (i = 0; i < root->nr_children; i++) {
			pid_child[i] = fork();
			if (pid_child[i] < 0) {
				perror("fork-error");
				exit(1);
			}
```

```
            if(pid_child[i] == 0) {
                fork_procs(root->children + i);
            }
        }

        wait_for_ready_children(root->nr_children);

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

        for (i=0; i < root->nr_children; i++) {
            kill(pid_child[i],SIGCONT);
            pid_child[i] = wait(&wait_status);
            if (pid_child[i] < 0) {
                perror("wait-error");
                exit(1);
            }
            explain_wait_status(pid_child[i], wait_status);
        }
        exit(12); //arbitrary value for non-leaf procs
    }
}

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);
```

```
        }

        root = get_tree_from_file(argv[1]);
        printf("Print tree:\n");
        print_tree(root);
        printf("\n");

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

        wait_for_ready_children(1);
        show_pstree(pid);

        kill(pid, SIGCONT);
        wait(&status);
        explain_wait_status(pid, status);
        return 0;
}
```

Β) Η έξοδος εκτέλεσης του προγράμματος

Η έξοδος που παίρνουμε για το αρχείο proc.tree είναι:

```
oslab04@orion:~/ex2/forktree3$ ./ask2-signals proc.tree
Print tree:
A
        B
                E
                F
        C
        D

PID = 21441, name A, starting...
PID = 21444, name D, starting...
My PID = 21441: Child PID = 21444 has been stopped by a signal, signo = 19
PID = 21442, name B, starting...
PID = 21443, name C, starting...
My PID = 21441: Child PID = 21443 has been stopped by a signal, signo = 19
PID = 21446, name F, starting...
My PID = 21442: Child PID = 21446 has been stopped by a signal, signo = 19
PID = 21445, name E, starting...
My PID = 21442: Child PID = 21445 has been stopped by a signal, signo = 19
My PID = 21441: Child PID = 21442 has been stopped by a signal, signo = 19
My PID = 21440: Child PID = 21441 has been stopped by a signal, signo = 19


A(21441)─┬─B(21442)─┬─E(21445)
         │          └─F(21446)
         ├─C(21443)
         └─D(21444)


PID = 21441, name = A is awake
PID = 21442, name = B is awake
PID = 21445, name = E is awake
My PID = 21442: Child PID = 21445 terminated normally, exit status = 15
PID = 21446, name = F is awake
My PID = 21442: Child PID = 21446 terminated normally, exit status = 15
My PID = 21441: Child PID = 21442 terminated normally, exit status = 12
PID = 21443, name = C is awake
My PID = 21441: Child PID = 21443 terminated normally, exit status = 15
PID = 21444, name = D is awake
My PID = 21441: Child PID = 21444 terminated normally, exit status = 15
My PID = 21440: Child PID = 21441 terminated normally, exit status = 12
oslab04@orion:~/ex2/forktree3$
```

Χρησιμοποιούμε το Makefile ως έχει.

Γ) Σύντομες απαντήσεις στις ερωτήσεις (1,2)

1) Με τη χρήση σημάτων, αντί της sleep() για τον συγχρονισμό των διεργασιών, προσφέρεται αρχικά το πλεονέκτημα του χρόνου, αφού δε χρειάζεται κάθε φορά να περιμένουμε να καλείται η sleep(). Επίσης, ενώ στο 1.2 μπορούσαμε να έχουμε εκτύπωση μηνυμάτων δημιουργίας διεργασιών μόνο με σειρά BFS, με τα σήματα στο 1.3, μπορούμε να έχουμε και DFS (ντετερμινιστική σειρά, δεν εξαρτάται από τον scheduler)

2) Η συνάρτηση wait_for_ready_children αναμένει μέχρι όλα τα children της διεργασίας που την κάλεσε να σταματήσουν. Με αυτό τον τρόπο εξασφαλίζεται η DFS σειρά εκτύπωσης των μηνυμάτων. Χωρίς αυτή η σειρα δεν θα ήταν ντετερμινιστική και θα εξαρτώταν από τον scheduler.

1.4 Σύνδεση με αρχείο αντικειμένων

Α) Ο πηγαίος κώδικας (source code) της άσκησης

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <string.h>
#include "tree.h"
#include "proc-common.h"

void fork_procs(struct tree_node *node, int pipe_fd) {
    int wait_status, pfd[2];
    int children = node->nr_children;
    int content = atoi(node->name);
    int content2[2];
    pid_t pid[children];
    if (node == NULL) {
        perror("node-error");
        exit(1);
    }
    change_pname(node->name);
    if (children == 0) {
        if (write(pipe_fd, &content, sizeof(content)) != sizeof(content)){
            perror("pipe-error: write to pipe");
            exit(1);
        }
        close(pipe_fd);
        raise(SIGSTOP);
        exit(17); //value is arbitrary
    }
    else {
```

```
if (pipe(pfd) < 0) {
        perror("pipe-error");
        exit(1);
}
int i;
for (i=0; i < children; i++) {
        pid[i] = fork();
        if (pid[i] < 0) {
                perror("fork-error");
                exit(1);
        }
        if (pid[i] == 0) {
                close(pfd[0]);
                fork_procs(node->children+i, pfd[1]);

        }
}
close(pfd[1]);
for (i=0; i < children; i++) {
        if (read(pfd[0], &content2[i], sizeof(content2[i])) != sizeof(content2[i])) {
                perror("pipe-error: read from pipe");
                exit(1);
        }
}
close(pfd[0]);
int result;
switch(node->name[0]) {
        case '+':
                result = content2[0] + content2[1];
                break;
        case '*':
                result = content2[0] * content2[1];
```

```
                    break;
                default:
                    break;
            }
            printf("%ld, Calculation:  %i %s %i = %i\n", (long)getpid(), content2[0], node->name,
content2[1], result);
            if (write(pipe_fd, &result, sizeof(result)) != sizeof(result)) {
                perror("pipe-error: write to pipe");
                exit(1);
            }
            close(pipe_fd);
            raise(SIGSTOP);
            for (i=0; i < children; i++) {
                kill(pid[i],SIGCONT);
                pid[i] = wait(&wait_status);
                if (pid[i] < 0) {
                    perror("wait-error");
                    exit(1);
                }
                explain_wait_status(pid[i], wait_status);
            }
            exit(12);
        }
}
int main(int argc, char **argv)
{
    int something;
    int pfd[2]; /*from pipe-example.c*/
    pid_t pid;
    int status;
    struct tree_node *root;
```

```
if (argc != 2) {
	printf("Usage: ./<file> <tree_file>\n");
	exit(1);
}

root = get_tree_from_file(argv[1]);

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

printf("Parent: Creating child...\n");
pid = fork();
if (pid < 0) {
	/* fork failed */
	perror("fork-error");
	exit(1);
}
if (pid == 0) {
	/* In child process */
	close(pfd[0]);
	fork_procs(root,pfd[1]);
}
close(pfd[1]); //we dont write...
if (read(pfd[0], &something, sizeof(something)) != sizeof(something)) {
	perror("10 pipe-error: read from pipe");
	exit(1);
}
close(pfd[0]);
show_pstree(pid);
```

```
        kill(pid, SIGCONT); /* We use signals, not sleep() */

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

        printf("The result of the expression is: %i\n", something);
        return 0;
}
```

Β) Η έξοδος εκτέλεσης του προγράμματος

Η έξοδος του προγράμματος για είσοδο expr.tree είναι:

```
oslab04@orion:~/ex2/forktree4$ ./ask2-calc expr.tree
Parent: Creating pipe...
Parent: Creating child...
22224, Calculation:  5 + 7 = 12
22223, Calculation:  4 * 12 = 48
22221, Calculation:  10 + 48 = 58


+(22221)─┬─*(22223)─┬─+(22224)─┬─5(22226)
         │          │          └─7(22227)
         │          └─4(22225)
         └─10(22222)


My PID = 22221: Child PID = 22222 terminated normally, exit status = 17
My PID = 22224: Child PID = 22226 terminated normally, exit status = 17
My PID = 22224: Child PID = 22227 terminated normally, exit status = 17
My PID = 22223: Child PID = 22224 terminated normally, exit status = 12
My PID = 22223: Child PID = 22225 terminated normally, exit status = 17
My PID = 22221: Child PID = 22223 terminated normally, exit status = 12
My PID = 22220: Child PID = 22221 terminated normally, exit status = 12
The result of the expression is: 58
oslab04@orion:~/ex2/forktree4$
```

Υλοποιείται η πράξη 10 + (4 * (7 + 5)) = 58.

Γ) Σύντομες απαντήσεις στις ερωτήσεις (1,2)

1) Καθώς χρησιμοποιούμε μόνο τους τελεστές “*” και “+”, που ορίζουν πράξεις αντιμεταθετικές, δεν μας ενδιαφέρει η σειρά με την οποία θα εκτελεστούν οι πράξεις, οπότε μας αρκεί μία μόνο σωλήνωση. Δηλαδή έχουμε μία σωλήνωση συνολικά για κάθε leaf-process και 2 σωληνώσεις για κάθε non-leaf process (η μία δέχεται τις τιμές των child-processes της και η άλλη δίνει τιμές στην parent-process). Αν επιθυμούσαμε να έχουμε και αφαιρέσεις ή διαιρέσεις, τότε ,καθώς είναι μη αντιμεταθετικές πράξεις, θα χρειαζόμασταν 2 pipes για κάθε leaf-process και 3 για κάθε non-leaf process.
2) Σε ένα σύστημα πολλαπλών επεξεργαστών, μπορούν να εκτελούνται παραπάνω από μια διεργασίες παράλληλα, οπότε η αποτίμηση της έκφρασης από δέντρο διεργασιών έχει πλεονέκτημα έναντι της αποτίμησης από μία μόνο διεργασία. Αυτό οφείλεται στο ότι είναι δυνατός ο παράλληλος υπολογισμός υποτμημάτων του αρχικού δέντρου. Συγκεκριμένα, πράξεις ίδιου επιπέδου αλλά διαφορετικού κλάδου μπορούν να εκτελούνται παράλληλα.