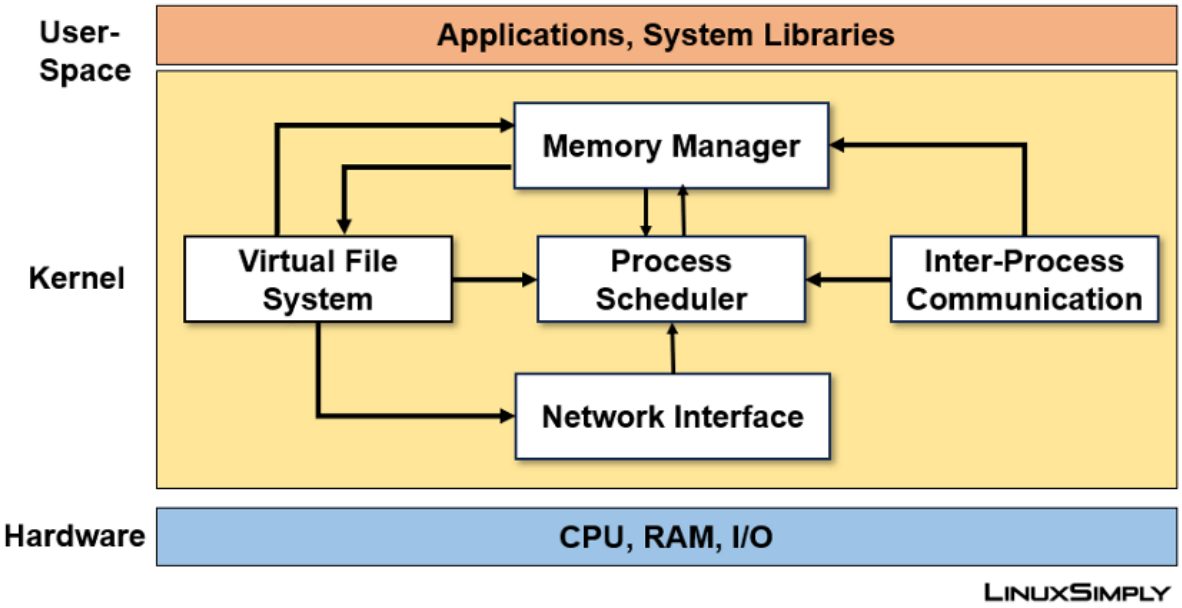
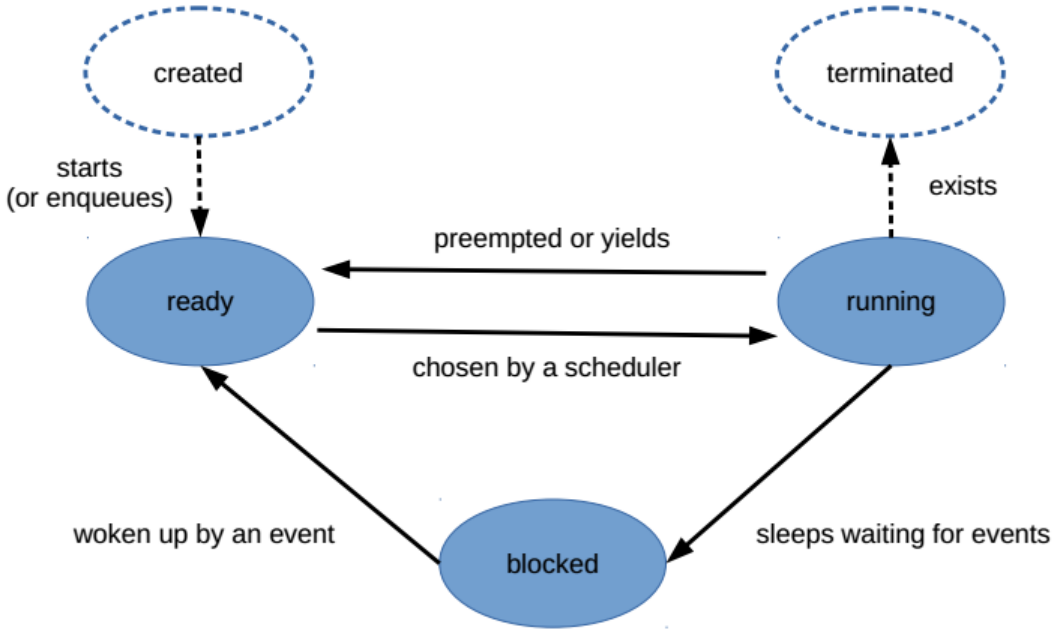
Υλοποίηση Αλγορίθμου Χρονοπρογραμματισμού (Task Scheduler) στον Linux Kernel (Έκδοση 6.14.8)

Λαζανάς Αλέξιος lazanas.ale@gmail.com

July 2, 2025





Contents

0 Σχόλια και Παραδοχές

- Για την υλοποίηση του πρότζεκτ έχει χρησιμοποιηθεί η έκδοση 6.14.8 του linux kernel
- Για την οπτικοποίηση των στοιχείων κάθε διεργασίας όταν εκείνη τελειώνει σε ραβδογράμματα, τα στοιχεία της διεργασίας αντιγράφονται από το debug pipe (/sys/kernel/debug/tracing/trace) σε csv αρχείο μέσω του shell script process_data_visualization/copy_to_csv.sh.
- Το προαναφερθέν csv διαβάζεται από ένα python script που μέσω της matplotlib εμφανίζει τα κατάλληλα ραβδογράμματα. Για την συγγραφή του κώδικα του python script, ευχαριστώ πολύ για την συνεισφορά της την Σταματία Καλημέρη (stamykalim@gmail.com).
- Από την έκδοση 6.11 και έπειτα η συγγραφή και η ενσωμάτωση scheduling αλγορίθμων στο linux kernel είναι ευκολότερη και ασφαλέστερη μέσω της προσθήκης της extensible_sched_class και πλέον μπορεί κανείς να προσθέσει schedulers μέσα από loadable bpf modules. Στο συγκεκριμένο πρότζεκτ επέλεξα να ακολουθήσω την "old school" μέθοδο επεκτείνοντας τον κώδικα του kernel απ' ευθείας, μπορεί αυτή η μέθοδος να απαιτεί μεγαλύτερο κόπο, αλλά θεωρώ ότι δίνει πιο στέρεη εκπαιδευτική εμπειρία.

1 Περιγραφή του Αλγορίθμου Χρονοπρογραμματισμού

Ο Αλγόριθμος χρονοπρογραμματισμού που υλοποίησα είναι πολύ απλός. Κάθε διεργασία έχει δύο dealines D1 και D2 με D1 < D2 και εκτιμώμενο χρόνο εκτέλεσης X ο οποίος πρέπει να μπορεί να ολοκληρωθεί εντός των deadlines. Τα τρία πεδία αυτά προσδίδονται στην τρέχουσα διεργασία μέσω μιας νέας κλήσης συστήματος (system call) που όρισα. Στην συνέχεια ο scheduler υπολογίζει το σκορ της διεργασίας με βάσει αυτά τα τρία πεδία ως εξής: Αν ο εκτιμώμενος χρόνος εκτέλεσης δείχνει ότι η διεργασία θα τελειώσει πριν την χρονική στιγμή D1 τότε το σκορ είναι 100, αντίθετα αν μας πληροφορεί ότι θα τελειώσει μετά από την D2 τότε το σκορ είναι 0. Αν μας πληροφορεί ότι θα τελειώσει μεταξύ των χρονικών στιγμών D1 και D2 υπολογίζεται από τον τύπο: $\frac{D2-X}{D2-D1} * 100$.

Ο scheduler θα επιλέξει για να τρέξει την διεργασία με το μέγιστο σκορ και θα της δώσει χρονομερίδιο ανάλογο του σκορ της. Μετά από την λήξη του χρονομεριδίου το σκορ μειώνεται με ρυθμό ανάλογο του χρόνου που έχει καταναλώσει στην cpu μέχρι στιγμής. Αν η διεργασία ξεπεράσει ως χρόνο εκτέλεσης εκείνον που δήλωσε στα system calls της δίνεται πέναλτι μειώνοντας το σκορ κατά το τετράγωνο της διαφοράς του χρόνου που έχει τρέξει με τον χρόνο που υποσχέθηκε ότι θα τρέξει.

Η προεκτόπιση (preemption) γίνεται με την λήξη του χρονομεριδίου. Σε περίπτωση που η τρέχουσα διεργασία έχει ξεπεράσει τον χρόνο που υποσχέθηκε και μια άλλη μόλις έγινε runnable από μπλοκαρισμένη η οποία έχει μεγαλύτερο σκορ τότε γίνεται επίσης προεκτόπιση της τρέχουσας διεργασίας.

2 Λίγα Λόγια για την δομή του Χρονοπρογραμματιστή στον Linux Kernel

2.1 Αντικειμενοστρέφεια και Κληρονομικότητα στην C;

Ο τίτλος της υποπαραγράφου ίσως να φαίνεται πρωτοφανείς καθώς όλοι γνωρίζουμε ότι η γλώσσα C υποστηρίζει διαδικαστικό προγραμματισμό, όχι αντικειμενοστρεφή.

Ας δούμε ένα παράδειγμα κληρονομικότητας σε μια γνωστή αντικειμενοστρεφή γλώσσα όπως η Java.

```
abstract class ExampleClass {
    public abstract int someMethod(int someArgument);
}

public class Main {
    public static void main(String[] args){
        ExampleClass exampleInstance = new ExampleClass {
            public int someMethod(int someArgument) {
                System.out.println("Here_is_my_argument:"+someArgument);
            }
        }
    }
}
```

Το παραπάνω παράδειγμα κληρονομικότητας πρακτικά είναι το μοναδικό που μπορεί να μεταφερθεί αυτούσιο στην γλώσσα C ως εξής:

```
struct ExampleClass {
    int (*someMethod)(int someArgument);
};

int someMethod_impl(int someArgument){
    printf("Here_is_my_argument:%d\n",someArgument);
}

struct ExampleClass exampleInstance = {
    .someMethod = someMethod_impl
};
```

Ο linux kernel είναι γραμμένος σε GNU C (επειδή για αρκετά low level parts απαιτείται να εισάγουμε inline assembly που μόνο το GNU C πρότυπο επιτρέπει). Όμως σε κάποια μέρη, όπως η δομή των character αρχείων και οι χρονοπρογρμματιστές, με τους οποίους ασχολήθηκα στο παρόν πρότζεκτ, είναι χρήσιμη μια πιο αντικειμενοστρεφής προσέγγιση και συμπεριφορά. Με τον τρόπο που μόλις εξηγήθηκε ο kernel του linux ενσωματώνει την αντικειμενοστρεφή προσέγγιση στον κώδικά του.

Η δομή αυτή που ακολουθείται στους χρονοπρογραμματιστές μας δίνει την δυνατότητα να επεκτείνουμε και να δημιουργήσουμε νέους αλγορίθμους scheduling με αρκετή ευκολία και μικρή επέμβαση στον κώδικα του linux kernel.

2.2 Η "Αφηρημένη Κλάση" του Χρονοπρογραμματιστή στο Linux Kernel

Έχοντας αντιληφθεί πλέον πως μπορούμε να εξομοιώσουμε την αντικειμενοστρεφή προσέγγιση στην γλώσσα C, είναι πλέον κατάλληλο να ρίξουμε μια ματιά στον κώδικα της αφηρημένης κλάσης του χρονοπρογραμματιστή (**struct** sched_class)

```
struct sched_class {

#ifdef CONFIG_UCLAMP_TASK
        int uclamp_enabled;
#endif

        void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
        bool (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
        void (*yield_task)   (struct rq *rq);
        bool (*yield_to_task)(struct rq *rq, struct task_struct *p);

        void (*wakeup_preempt)(struct rq *rq, struct task_struct *p, int flags);

        int (*balance)(struct rq *rq, struct task_struct *prev, struct rq_flags *rf);
        struct task_struct *(*pick_task)(struct rq *rq);
        /*
         * Optional! When implemented pick_next_task() should be equivalent to:
         *
         *   next = pick_task();
         *   if (next) {
         *       put_prev_task(prev);
         *       set_next_task_first(next);
         *   }
         */
        struct task_struct *(*pick_next_task)(struct rq *rq, struct task_struct *prev);

        void (*put_prev_task)(struct rq *rq, struct task_struct *p, struct task_struct *next);
        void (*set_next_task)(struct rq *rq, struct task_struct *p, bool first);

#ifdef CONFIG_SMP
        int  (*select_task_rq)(struct task_struct *p, int task_cpu, int flags);

        void (*migrate_task_rq)(struct task_struct *p, int new_cpu);

        void (*task_woken)(struct rq *this_rq, struct task_struct *task);

        void (*set_cpus_allowed)(struct task_struct *p, struct affinity_context *ctx);

        void (*rq_online)(struct rq *rq);
        void (*rq_offline)(struct rq *rq);

        struct rq *(*find_lock_rq)(struct task_struct *p, struct rq *rq);
#endif

        void (*task_tick)(struct rq *rq, struct task_struct *p, int queued);
        void (*task_fork)(struct task_struct *p);
        void (*task_dead)(struct task_struct *p);

        /*
         * The switched_from() call is allowed to drop rq->lock, therefore we
         * cannot assume the switched_from/switched_to pair is serialized by
         * rq->lock. They are however serialized by p->pi_lock.
         */
        void (*switching_to) (struct rq *this_rq, struct task_struct *task);
        void (*switched_from)(struct rq *this_rq, struct task_struct *task);
        void (*switched_to)  (struct rq *this_rq, struct task_struct *task);
        void (*reweight_task)(struct rq *this_rq, struct task_struct *task,
                              const struct load_weight *lw);
        void (*prio_changed) (struct rq *this_rq, struct task_struct *task,
                              int oldprio);

        unsigned int (*get_rr_interval)(struct rq *rq,
                                        struct task_struct *task);

        void (*update_curr)(struct rq *rq);

#ifdef CONFIG_FAIR_GROUP_SCHED
        void (*task_change_group)(struct task_struct *p);
#endif

#ifdef CONFIG_SCHED_CORE
        int (*task_is_throttled)(struct task_struct *p, int cpu);
#endif
};
```

Κάθε αλγόριθμος χρονοπρογραμματισμού που υπάρχει στο linux kernel είναι στην ουσία αντικείμενο αυτής της κλάσης που υλοποιεί μεγάλο μέρος αυτών των μεθόδων, δεν είναι απολύτως απαραίτητο να υλοποιηθούν όλες, κάποιοι από αυτούς τους pointers είναι πιθανό να παραμείνουν NULL σε κάποια instances.

2.3 Πως αποθηκεύονται οι διεργασίες και οι διαφορετικοί Χρονοπρογραμματιστές;

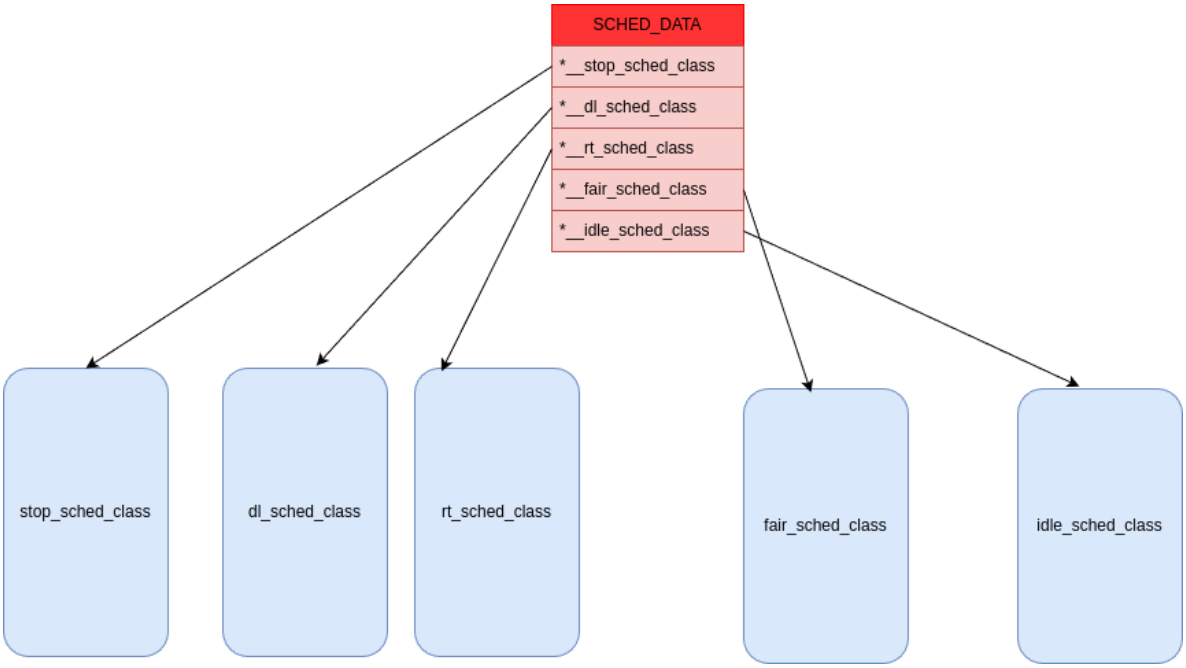
2.3.1 Δομή της task_struct

Στο linux kernel υπάρχει η struct task_struct που στην ουσία ορίζει την δομή μιας διεργασίας. Είναι πρακτικά πάρα πολύ μεγάλη σε χώρο struct, για αυτό τον λόγο δεν αποθηκεύεται εκείνη στην ουρά αλλά μόνο το μέρος της που είναι απαραίτητο για το scheduling και μέσω του macro container_of() του linux kernel μπορούμε από έναν pointer σε ένα μέλος ενός αντικειμένου να επιστρέψουμε έναν pointer στο ευρύτερο αντικείμενο.

Η struct αυτή περιέχει όλα τα δεδομένα που υπάρχουν για μια διεργασία ή ένα thread (το Linux δεν έχει ξεχωριστή δομή για threads). Είναι πραγματικά τεράστια σε χώρο και κώδικα, χρησιμοποιείται μόνο όταν είναι απολύτως απαραίτητη και μόνο μέσω pointers σε αντικείμενά της. Τα αντικείμενα τας task_struct είναι αποθηκευμένα στο kernel heap memory, λόγω του μικρού μεγέθους της kernel stack στην στοίβα αποθηκεύονται μόνο pointers σε αντικείμενα τα οποία, αυτούσια είναι στον heap.

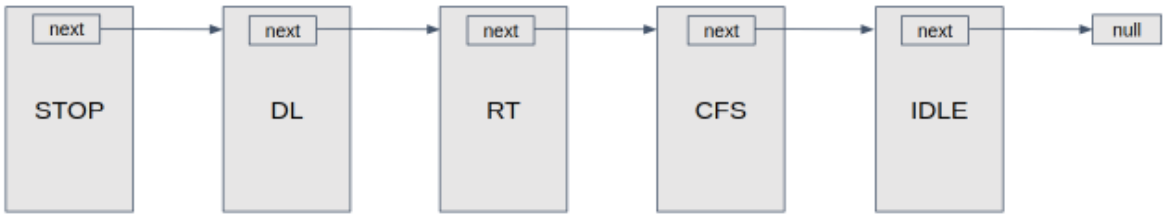
2.3.2 Η αποθήκευση των Χρονοπρογραμματιστών

Το linux έχει από μόνο του 5 instances της κλάσης sched_class υλοποιώντας 5 διαφορετικούς scheduling αλγόριθμους για 5 διαφορετικές κατηγορίες διεργασιών. Τα 5 αυτά instances αποθηκεύονται στο kernel-data-memory (ως καθολικές μεταβλητές) και ένα array από pointers σε αυτά αποθηκεύεται στην kernel-stack. Στο array αυτό οι pointers στους schedulers είναι αποθηκευμένοι με την σειρά που δείχνει η παρακάτω εικόνα:



οι pointers __sched_class_highest και __sched_class_lowest είναι στην ουσία ίσοι με SCHED_DATA και SCHED_DATA+4 αντίστοιχα.

Σε παλιότερες εκδόσεις του linux kernel η struct sched_class είχε ένα ακόμα data member τύπου (struct sched_class *next) που και οι διαφορετκοί schedulers αποθήκευαν οι ίδιοι σε συνδεδεμένη λίστα την σειρά με την οποία θα έτρεχαν όπως παρακάτω:



Αλλά αυτό για λόγους efficiency άλλαξε από την έκδοση 5.7 και έπειτα και μετάβηκαν στο μοντέλο που εξήγησα παραπάνω.

2.4 Επιλογή του επόμενου task

Η βασική scheduling λογική βρίσκεται στην συνάρτηση __pick_next_task στο αρχείο kernel/sched/-core.c. Πρακτικά διατρέχουμε όλο το array με τους διαφορετικούς schedulers και καλούμε την συνάρτηση pick_next_task() για καθέναν από αυτούς (που όπως δείξαμε παραπάνω είναι μια συνάρτηση που υλοποιούν όλοι οι schedulers).

Από προεπιλογή όλες οι διεργασίες αν δεν διαλέξουν οι ίδιες τον χρονοπρογραμματιστή ο οποίος θα τις διαχειρίζεται αωατίθενται στο instance fair_sched_class που είναι ο default χρονοπρογραμματιστής του linux. Συνεπώς υπάρχει ένα optimization ότι αν όλες οι τρέχουσες διεργασίες είναι στην fair_sched_class να μην κοιτάξει καν τις υπόλοιπες και να τρέξει εκείνη που θα κάνει schedule η fair.

```
static inline struct task_struct *
__pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
        const struct sched_class *class;
        struct task_struct *p;

        rq->dl_server = NULL;

        if (scx_enabled())
                goto restart;

        /*
         * Optimization: we know that if all tasks are in the fair class we can
         * call that function directly, but only if the @prev task wasn't of a
         * higher scheduling class, because otherwise those lose the
         * opportunity to pull in more work from other CPUs.
         */
        if (likely(!sched_class_above(prev->sched_class, &fair_sched_class) &&
                   rq->nr_running == rq->cfs.h_nr_queued)) {

                p = pick_next_task_fair(rq, prev, rf);
                if (unlikely(p == RETRY_TASK))
                        goto restart;

                /* Assume the next prioritized class is idle_sched_class */
                if (!p) {
                        p = pick_task_idle(rq);
                        put_prev_set_next_task(rq, prev, p);
                }

                return p;
        }

restart:
        prev_balance(rq, prev, rf);

        for_each_active_class(class) {
                if (class->pick_next_task) {
                        p = class->pick_next_task(rq, prev);
                        if (p)
                                return p;
                } else {
                        p = class->pick_task(rq);
                        if (p) {
                                put_prev_set_next_task(rq, prev, p);
                                return p;
                        }
                }
        }

        BUG(); /* The idle class should always have a runnable task. */
}
```

3 Διαλέγοντας τον Χρονοπρογραμματιστή για μια Διεργασία

Όπως αναφέρθηκε νωρίτερα μια διεργασία έχει την δυνατότητα καθώς τρέχει να διαλέξει τον χρονοπρογραμματιστή στον οποίο θα ανατεθεί καθώς και να τον αλλάζει κατ' απαίτηση. Όλες οι διεργασίες έχουν έναν default χρονοπρογραμματιστή με την πλειοψηφία αυτών να έχει τον fair ως προεπιλογή. Πώς όμως μπορεί μια διεργασία να διαλέξει τον scheduler στον οποίο θα ανατεθεί στην συνέχεια και να τον αλλάξει on demand?

Το linux παρέχει ήδη μια κλήση συστήματος για αυτή την λειτουργία η οποία είναι δηλωμένη για C user-space προγράμματα στο header file include uapi/linux/sched.h και μπορεί οποιοδήποτε C πρόγραμμα να συμπεριλάβει αυτό το Header ως εξής #include <linux/sched.h>. Η κλήση συστήματαος αυτή είναι η sched_setscheduler(), η οποία παίρνοντας όρισμα το pid της επιθυμητής διεργασίας (αν είναι 0 λειτουργεί για την διεργασία που κάλεσε το system call), την σταθερά που καθορίζει τον αλγόριθμο χρνοπρογραμματισμού (SCHED_FIFO (αναθέτει στον rt_sched_class με άπειρο χρονομερίδιο(Αλγόριθμος FCFS)), SCHED_RR (αναθέτει στον rt_sched_class με περιορισμένο κβάντο χρόνου (Αλγόριθμος Round Robin)), SCHED_NORMAL(αναθέτει στον fair_sched_class (Αλγόριθμος CFS (DEFAULT))) και εναν δέικτη σε ένα sched_attribute (μπορεί να είναι NULL) αλλάζει τον scheduler της διεργασίας και την αναθέτει στον επιθυμητό.

Εγώ αφού όρισα την δική μου σταθερά (SCHED_HVF) έκανα την αντιστοίχιση στην κλάση hvf_sched_class και στην ουσία στο linux σύστημα που έχει τον custom kernel που έφτιαξα είναι ικανό να αντιστοιχίσει μια οποιαδήποτε διεργασία και στον δικό μου scheduler βάζοντας ως όρισμα την σταθερά που όρισα (τιμή 8).

4 Η υλοποίηση της νέας πολιτικής στον Linux kernel

Η υλοποίηση ουσιατικά του νέου scheduling αλγορίθμου απαιτούσε αρκετές φορές την προσθήκη κώδικα σε ήδη υπάρχουσες συναρτήσεις και structs στο Linux kernel. Όλα τα αρχεία που χρειάστηκε να επεξεργαστώ κατα την υλοποίηση βρίσκονται στο github repo που έφτιαξα. Προφανώς στο παρόν έγγραφο θα αναφερθώ μόνο στα κομμάτια κώδικα που είναι απολύτως βασικά για την υλοποίηση του αλγορίθμου μου όχι ιδιαίτερα για την καταγραφή και την ενσωμάτωσή του στο linux kernel, καθώς αυτός ο κώδικας είναι πιο πολύ κοπτοραπτική και θα έλεγα ότι αν κάποιος θέλει να φτιάξει ένα αντίστοιχο πρότζεκτ αυτό είναι το εύκολο μέρος του.

4.1 Επιπρόσθετες κλήσεις συστήματος (system calls)

Για την υλοποίηση του scheduler προστέθηκαν 4 ακόμα πεδία στην task_struct (αυτά που συζητήθηκαν στην αρχή του αλγορίθμου + μια boolean τιμή που καθορίζει αν τα πεδία αυτά έχουν καθοριστεί από τον χρήστη (μέσα από την κλήση συστήματος που δημιούργησα)

```c
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/my_sched_sys.h>
#include <asm/current.h>
#include <asm-generic/errno.h>
#include <linux/syscalls.h>
#include <linux/uaccess.h>
#include <linux/types.h>
#include <linux/timekeeping.h>


#define M 1000
#define H 100

SYSCALL_DEFINE3(set_scheduling_params, long, deadline_1, long, deadline_2, long, computation_time){
    bool condition_1 = (deadline_1>0 && deadline_2>0) && deadline_1<deadline_2;
    struct timespec64 now;
    ktime_get_real_ts64(&now);

    bool condition_2 = (now.tv_sec*M+computation_time)<deadline_2*M;

    if (condition_1 && condition_2){
        current->deadline_1 = deadline_1;
        current->deadline_2 = deadline_2;
        current->computation_time = computation_time;
        current->pars_set = true;

        return 0;
    }
    return -EINVAL;
}


SYSCALL_DEFINE1(get_scheduling_params, struct d_params *, params){
    bool condition = (params!=NULL) && access_ok(params, sizeof(struct d_params));
    if (condition){
        if (copy_to_user(&params->deadline_1, &current->deadline_1, sizeof(long))!=0){
            return -EFAULT;
        }
        if (copy_to_user(&params->deadline_2, &current->deadline_2, sizeof(long))!=0){
            return -EFAULT;
        }
        if (copy_to_user(&params->computation_time, &current->computation_time, sizeof(long))!=0){
            return -EFAULT;
        }
        return 0;
    }
    return -EINVAL;
}
```

Με το πρώτο system call η τρέχουσα διεργασία καθορίζει τις παραμέτους της ενώ με το δεύτερο επιστρέφονται οι παράμετροί της. Έγραψα και ένα τρίτο system call που επιστρέφει το σκορ που θα είχε η διεργασία αν τελείωνε την στιγμή που κάλεσε αυτό το syscall

```c
SYSCALL_DEFINE0(get_scheduling_score){
    struct timespec64 now;
    ktime_get_real_ts64(&now);
    long D1 = current->deadline_1;
    long D2 = current->deadline_2;
    long D3 = now.tv_sec;

    if (D3<D1)
        return H;
    else if (D2<D3)
        return 0;
    else
        return ((D2-D3)*100)/(D2-D1);
}
```

4.2 Enquing and dequing a Task

Σύμφωνα με την πολιτική όπως αναλύθηκε στην αρχή του παρόντος εγγράφου ο scheduler θα επιλέξει πάντα την διεργασία με το μέγιστο σκορ για να τρέξει. Οπότε θα μας ήταν χρήσιμη μια δομή δεδομένων που γρήγορα έχει την ικανότητα να επιστρέψει το task με το μέγιστο σκορ, μια τέτοια δομή θα ήταν ο σωρός (heap) όμως ο σωρός είναι inefficient σε arbitary deletion σε περίπτωση που ένα task μπλοκάρει λόγω I/O ή λόγω κλήσης της sleep. Συνεπώς επέλεξα μια δομή δεδομένων που είναι ικανή να εκτελέσει αποδοτικά και τις δυο αυτές λειτουργίες (σε $O(\log n)$ χρόνο) ένα υψοζυγισμένο δέντρο δυαδικής αναζήτησης (balanced binary tree) και συγκεκριμένα ένα red-black tree καθώς οι εισαγωγές και οι διαγραφές είναι πιο συχνές σε ένα scheduling context και τις χειρίζεται πιο αποδοτικά από την εναλλακτική του (το AVL Tree).

Ο linux kernel διαθέτει ένα ήδη ορισμένο header file με την υλοποίηση ενός red-black tree στο include/linux/rbtree.h αυτό το red-black tree χρησιμοποίησα για την υλοποίηση της runqueue του αλγορίθμου χρονοπρογραμματισμού μου. Στο δέντρο αποθηκεύω αντικείμενα τύπου sched_hvf_entity με κριτήριο το scheduling score τους.

Παρακάτω βρίσκονται όλες οι συναρτήσεις που συμβάλλουν στην εισαγωγή ενός sched_hvf_entity στην runqueue του χρονοπρογραμματιστή μου.

```
static void
enqueue_task_hvf(struct rq *rq, struct task_struct *p, int flags){
    struct hvf_rq *hvf_rq = &rq->hvf;
    struct sched_hvf_entity *se_hvf = &p->hvf;

    if (flags & (ENQUEUE_INITIAL | ENQUEUE_CHANGED)){
        init_sched_hvf_entity(se_hvf);
        compute_init_sched_value(p);
    }
    else if (flags & (ENQUEUE_WAKEUP | ENQUEUE_MIGRATED | ENQUEUE_RESTORE)){
        compute_init_sched_value(p);
    }

    if (se_hvf != hvf_rq->curr && !se_hvf->on_rq){
        enqueue_hvf_entity(hvf_rq, se_hvf);
        rq->nr_running++;
    }
}
```

```
inline void init_sched_hvf_entity(struct sched_hvf_entity *se){
    struct timespec64 now;
    ktime_get_real_ts64(&now);
    se->first_time = now.tv_sec*K + now.tv_nsec/(K*K);
    se->latest_time = se->first_time;
    se->time_used = 0;
    se->cpu_answers = 0;
    se->slice_expired = false;
}
```

```
inline long compute_init_sched_value(struct task_struct *p){
    const long first_time = p->hvf.first_time;
    const long D1 = p->deadline_1*K;
    const long D2 = p->deadline_2*K;
    const long X = first_time+p->computation_time;
    const long V = (X<D1)? H : (D2<X)? 0 : (D2-X)*H/(D2-D1);

    p->hvf.init_sched_value = V;
    p->hvf.curr_sched_value = V;

    return V;
}
```

```c
static void enqueue_hvf_entity(struct hvf_rq *hvf_rq, struct sched_hvf_entity *se){
    struct task_struct *task_to_eq = task_hvf_of(se);

    if (exceeded_time(task_to_eq)){
        long ctime = task_to_eq->computation_time;
        penalty_hvf_entity(se, ctime);
    }

    hvf_rq_rbtree_insert(&hvf_rq->hvf_task_queue, se);
    hvf_rq->nr_hvf_queued++;
    se->on_rq = true;
    struct sched_hvf_entity *max_entity = hvf_rq->max_value_entity;

    if (!max_entity){
        hvf_rq->max_value_entity = rb_entry(rb_last(&hvf_rq->hvf_task_queue), struct sched_hvf_entity, run_node);
        return;
    }

    if (max_entity->curr_sched_value < se->curr_sched_value)
        hvf_rq->max_value_entity = rb_entry(rb_last(&hvf_rq->hvf_task_queue), struct sched_hvf_entity, run_node);
}
```

```c
inline long penalty_hvf_entity(struct sched_hvf_entity *se, long ctime){
    long diff = se->time_used - ctime;
    long old_value = se->curr_sched_value;
    long new_value = old_value - (diff*diff*old_value)/100;
    new_value = (new_value>0)? new_value : (old_value == 0)? 10: 0;

    se->curr_sched_value = new_value;
    return new_value;
}
```

```c
bool hvf_rq_rbtree_insert(struct rb_root *root, struct sched_hvf_entity *se){
    struct rb_node **new_node = &(root->rb_node), *parent = NULL;

    while (*new_node != NULL){
        struct sched_hvf_entity *this_entity = container_of(*new_node, struct sched_hvf_entity, run_node);
        int result = se->curr_sched_value - this_entity->curr_sched_value;

        parent = *new_node;
        if (result<=0)
            new_node = &((*new_node)->rb_left);
        else
            new_node = &((*new_node)->rb_right);
    }

    rb_link_node(&se->run_node, parent, new_node);
    rb_insert_color(&se->run_node, root);

    return true;
}
```

Πρακτικά όλες αυτές οι 6 συναρτήσεις υπολογίζουν τα διάφορα γνωρίσματα της κάθε διεργασίας και η hvf_rq_rbtree_insert εισάγει το sched_hvf_entity στην ουρά. Το αντικείμενο με την μέγιστη τιμή cacharetai για μεγαλύτερη αποδοτικότητα όταν το χρειαζόμαστε.

Όσον αφορά την εξαγωγή ενός entity από την ουρά οι συναρτήσεις που εμπλέκονται είναι οι παρακάτω:

```c
static bool
dequeue_task_hvf(struct rq *rq, struct task_struct *p, int flags){
    struct hvf_rq *hvf_rq = &rq->hvf;
    struct sched_hvf_entity *se_hvf = &p->hvf;

    if (se_hvf != hvf_rq->curr && se_hvf->on_rq){
        bool result = dequeue_hvf_entity(hvf_rq, se_hvf);
        rq->nr_running--;
        return result;
    }
    if (se_hvf == hvf_rq->curr && !task_is_running(p)){
        bool result = true;

        if (unlikely(se_hvf->on_rq))
            result = dequeue_hvf_entity(hvf_rq, se_hvf);

        rq->nr_running--;
        hvf_rq->curr = NULL;

        return result;
    }

    return false;
}
```

```c
static bool dequeue_hvf_entity(struct hvf_rq *hvf_rq, struct sched_hvf_entity *se){
    struct rb_node *max_node = &hvf_rq->max_value_entity->run_node;

    if (max_node == &se->run_node){
        struct rb_node *prev_node = rb_prev(&se->run_node);
        hvf_rq->max_value_entity = (prev_node!=NULL)? rb_entry(prev_node, struct sched_hvf_entity, run_node) : NULL;
    }

    rb_erase(&se->run_node, &hvf_rq->hvf_task_queue);
    RB_CLEAR_NODE(&se->run_node);
    hvf_rq->nr_hvf_queued--;
    se->on_rq = false;

    return true;
}
```

Το μοτίβο της εξαγωγής είναι ίδιο με εκείνο στην εισαγωγή και οι συναρτήσεις αυτές βεβαιώνουν πάντα ότι θα cachάρεται το entity με το μέγιστο σκορ. Είναι βελτιστοποιημένες ώστε να ενημερώνουν το entity με την μέγιστη τιμή μόνο όταν είναι χρήσιμο.

4.3 Επιλέγοντας το επόμενο Task

Για να διαλέξω το επόμενο task αρκεί απλά να επιστρέψω το task που αντιστοιχεί στο μέγιστο entity που έχω cacheάρει ως data member στην hvf_rq. Το γεγονός ότι κατά την εισαγωγή και την διαγραφή των tasks αποθηκεύω ξεχωριστά το μέγιστο entity με βοηθά να έχω διαθέσιμο το επόμενο task χωρίς να διατρέχω συνέχεια ολόκληρο το δέντρο για να το εντοπίσω.

Παρακάτω βρίσκονται όλες οι συναρτήσεις που εμπλέκονται στην επιλογή του επόμενου task.

```c
static struct task_struct *pick_next_task_hvf(struct rq *rq, struct task_struct *prev){
    if (prev->sched_class == &hvf_sched_class){
        struct sched_hvf_entity *prev_hvf = &prev->hvf;
        update_used_se_hvf(prev_hvf);
    }

    struct task_struct *next = pick_task_hvf(rq);
    if (!next)
        return NULL;
    else {
        prev->sched_class->put_prev_task(rq, prev, next);
        next->sched_class->set_next_task(rq, next, true);
    }
    return next;
}
```

```c
/*this function shall be called in pick_next_task for the previous task*/

inline void update_used_se_hvf(struct sched_hvf_entity *se){
    struct timespec64 now;
    ktime_get_real_ts64(&now);

    long current_time = now.tv_sec*K + now.tv_nsec/(K*K);
    se->time_used += current_time - se->latest_time;
}
```

```c
static struct task_struct *pick_task_hvf(struct rq *rq){
    struct sched_hvf_entity *se_hvf;
    struct hvf_rq *hvf_rq;

    hvf_rq = &rq->hvf;
    if (hvf_rq_empty(hvf_rq))
        return rq->idle;
    se_hvf = pick_entity_hvf(hvf_rq);
    return task_hvf_of(se_hvf);
}
```

```c
inline bool hvf_rq_empty(struct hvf_rq *hvf_rq){
    struct rb_root *root = &hvf_rq->hvf_task_queue;
    return RB_EMPTY_ROOT(root);
}
```

```c
static inline
struct sched_hvf_entity *pick_entity_hvf(struct hvf_rq *hvf_rq){
    return hvf_rq->max_value_entity;
}
```

Στην ουσία επιστρέφεται απλά η μέγιστη entity. Όταν όμως η ουρά είναι άδεια επιστρέφουμε το Idle task για να τοποθετήσουμε τον scheduler σε αναμονή ώστε να αναμένει την άφιξη ενός νέου προς εκτέλεση task (θα μπορούσαμε να επιστρέφουμε και NULL καθώς και σε αυτή την περίπτωση θα επέστρεφε το idle η idle_sched_class).

4.4 Ρυθμίζοντας το προηγούμενο task και καθορίζοντας το επόμενο task

Όπως είδαμε παραπάνω, η pick_next_task_hvf καλεί τις συναρτήσεις put_prev_task και set_next_task σε περίπτωση που επιστραφεί ένα έγκυρο task από την pick_task αυτές οι συναρτήσεις είναι υπεύθυνες να τοποθετήσουν το προηγούμενο task στην ουρά ξανά αν χρειάζεται (put_prev_task) και να βγάλουν από την ουρά το επόμενο και να το θέσουν ως τρέχον στην cpu (set_next_task).

Οι υλοποιήσεις των δυο αυτών συναρτήσεων στον scheduling αλγόριθμο που έφτιαξα είναι οι παρακάτω:

```
static void
set_next_hvf_entity(struct hvf_rq *hvf_rq, struct sched_hvf_entity *se){
    if (se->on_rq)
        dequeue_hvf_entity(hvf_rq, se);

    update_latest_se_hvf(se);

    hvf_rq->curr = se;
}


static void set_next_task_hvf(struct rq *rq, struct task_struct *p, bool first){
    struct sched_hvf_entity *se_hvf = &p->hvf;
    struct hvf_rq *hvf_rq = &rq->hvf;
    if (!first)
        return;

    set_next_hvf_entity(hvf_rq, se_hvf);
}
```

```
static void
put_prev_hvf_entity(struct hvf_rq *hvf_rq, struct sched_hvf_entity *se){
    if (!se->on_rq)
        enqueue_hvf_entity(hvf_rq, se);

    hvf_rq->curr = NULL;
}

static void put_prev_task_hvf(struct rq *rq, struct task_struct *prev, struct task_struct *next){
    struct sched_hvf_entity *se_hvf = &prev->hvf;
    struct hvf_rq *hvf_rq = &rq->hvf;

    if (task_is_running(prev)) {
        if (se_hvf->slice_expired){
            reduce_sched_value(se_hvf);
            se_hvf->slice_expired = false;
        }
        put_prev_hvf_entity(hvf_rq, se_hvf);
    }
}
```

4.5 Προεκτόπιση Διεργασιών

Ο αλγόριθμός μου βασίζεται σε δυναμικά χρονομερίδια όπου με την λήξη αυτών γίνεται η προεκτόπιση της τρέχουσας διεργασίας. Προεκτόπιση γίνεται απιπλέον και όταν γίνεται runnable μια διεργασία η οποία και η τρέχουσα έχει ξεπεράσει τον χρόνο που υποσχέθηκε, η τρέχουσα θα προεκτοπιστεί. Οι συναρτήσεις που καθορίζουν πότε θα γίνει η προεκτόπιση είναι οι: task_tick και wakeup_preempt.

Η συνάρτηση task_tick καλείται περιοδικά ανά την περίοδο του scheduler που είναι 1ms οπότε τα χρονομερίδια πρέπει κατά κανόνα να είναι της τάξης των millisecond.

```
static void task_tick_hvf(struct rq *rq, struct task_struct *curr, int queued){
    struct sched_hvf_entity *curr_ent = &curr->hvf;
    long slice = time_slice(curr_ent->curr_sched_value);

    curr_ent->runtime += TICK_NSEC;

    if (curr_ent->runtime >= slice*K*K){
        curr_ent->slice_expired = true;
        resched_curr(rq);
    }
}
```

Η κλήση της resched_curr είναι εκείνη που εκτελεί την προεκτόπιση της τρέχουσας διεργασίας σε περίπτωση που λήξει το χρονομερίδιο της.

Η συνάρτηση wakeup_preempt καλείται κάθε φορά όπου ένα νέο task μόλις έγινε runnable και μέσω αυτής γίνεται η δεύτερη περίπτωση προεκτόπισης που καθορίζει ο αλγόριθμος.

```
static void wakeup_preempt_hvf(struct rq *rq, struct task_struct *p, int flags){
    if (current->sched_class == &hvf_sched_class){
        struct sched_hvf_entity *curr_se_hvf = &current->hvf;
        struct sched_hvf_entity *p_hvf = &p->hvf;

        /*
         * preempt only if the current task has exceeded its time,
         * else preemption happens generally in time_slice expiration on task_tick_hvf
         */

        if (exceeded_time(current) && (p_hvf->curr_sched_value > curr_se_hvf->curr_sched_value))
            resched_curr(rq);
        }
    }
}
```

4.6 Ένα task τερματίζει

όταν μια διεργασία τερματίζει τότε καλείται η task_dead που είναι υπεύθυνη να διαγράψει όλες τις αναφορές αυτής της διεργασίας από την μνήμη. Στον δικό μου αλγόριθμο καταγράφει στο kernel debug trace pipe χρήσιμα στοιχεία για αυτήν και έπειτα διαγράφεται από όλες τις δομές δεδομένων στην μνήμη.

```
static void register_entity_info(struct sched_hvf_entity *se_hvf, int pid){
    long init_value = se_hvf->init_sched_value;
    struct timespec64 now;
    ktime_get_real_ts64(&now);
    long curr_time = now.tv_sec*K + now.tv_nsec/(K*K);

    long turnaround_time = curr_time - se_hvf->first_time;
    long wait_time = turnaround_time - se_hvf->time_used;
    long comp_time = se_hvf->time_used;
    long response_time = se_hvf->cpu_first_answer - se_hvf->first_time;


    trace_printk(
                    "hvf_task_terminated,%d,%ld,%ld,%ld,%ld,%ld\n",
                    pid,
                    init_value,
                    turnaround_time,
                    wait_time,
                    comp_time,
                    response_time
                );
}

static void task_dead_hvf(struct task_struct *p){
    struct rq *rq;
    struct rq_flags rf;
    struct sched_hvf_entity *se_hvf = &p->hvf;

    register_entity_info(se_hvf, p->pid);

    rq = task_rq_lock(p, &rf);

    struct hvf_rq *hvf_rq = &rq->hvf;
    if (se_hvf->on_rq)
        dequeue_hvf_entity(hvf_rq, se_hvf);
    hvf_rq->curr = NULL;

    task_rq_unlock(rq, p, &rf);
}
```

5 Τεστάροντας τον Χρονοπρογραμματιστή

Για να τεστάρω τον αλγόριθμο χρονοπρογραμματισμού μου χρειαζόμουν πρακτικά ένα testing περιβάλλον ασφαλές, ένα testbench πρόγραμμα που θα spawnάρει πολλές διεργασίες με διαφορετικά σκορ και θα τις αναθέτει στον δικό μου scheduler.

5.1 Testing Environment

Για το περιβάλλον πρακτικά χρησιμοποιήθηκε μια εικονική μηχανή που δημιουργήθηκε από το virtual machine manager με λειτουργικό σύστημα lubuntu 24.04 lts, έπειτα αντέγραψα το αρχείο .qcow2 της εικονικής μηχανής σε κάποιον από τους Home folders.

Το Linux έχει την δυνατότητα να μεταφραστεί σε ένα συμπιεσμένο kernel image μέσω της εντολής make bzImage. Αυτό το συμπεσμένο kernel image μπορώ να το χρησιμοποιήσω ώστε να bootάρω ένα ήδη υπάρχον linux vm με τον δικό μου ustom kernel μέσω της εξής εντολής:

```
[alexis-pc linux-testvm]# qemu-system-x86_64 -m 4096 -hda ubuntu24.04.qcow2 -kernel /home/alexis1/Kernels/linux-6.14.8-mod-exec/arch/x86/boot/bzImage
 -append "root=/dev/sda1 console=ttyS0" -nographic
```

με αυτό τον τρόπο bootάρουμε την εικονική μηχανή με την δική μας έκδοση του linux kernel. Μόλις τρέξουμε την εντολή έχουμε πλέον ανοίξει το τερματικό της εικονικής μηχανής από το τερματικό του υπολογιστή μας. Έτσι μπορούμε να εκτελούμε εντολές σε αυτό το σύστημα χωρίς να χρειάζεται η χρήση μιας πλήρους εικονικής μηχανής (αν και μπορούμε να κάνουμε και αυτό).

Για ευκολία μετέφερα το εκτελέσιμο αρχείο του testbench που περιγράφω στην συνέχεια το οποίο έγινε compiled με το flag static ώστε το εκτελέσιμο να είναι πλήρης και να μην απαιτείται δυναμική φόρτωση μεθόδων.

5.2 Testbench Used

Το testbench Που χρησιμοποιήθηκε είναι ένα απλό testbench που δημιουργεί 20 διεργασίες με διαφορετικές παραμέτρους η καθεμία και τις αναθέτει να χρονοπρογραμματιστούν από τον δικό μου scheduler.

```c
#include <stdio.h>
#include <math.h>
#include <unistd.h>
#include <stdlib.h>
#include <sched.h>
#include <time.h>
#include <sys/syscall.h>
#include <sys/wait.h>

#define DELAY 750
#define K 1000

#define SCHED_HVF 8

#define __NR_SET_SCHED_PARS     467
#define __NR_GET_SCHED_PARS     468
#define __NR_GET_SCHED_SCORE    469

#define set_sched_params(x, y, z)   syscall(__NR_SET_SCHED_PARS, x, y, z)
#define get_sched_params(x)         syscall(__NR_GET_SCHED_PARS, x)


double a = 1.1;

void core_delay(){
    unsigned long j;

    for (j = 0; j < 100000; j++) {
        a += sqrt(1.1)*sqrt(1.2)*sqrt(1.3)*sqrt(1.4)*sqrt(1.5);
        a += sqrt(1.6)*sqrt(1.7)*sqrt(1.8)*sqrt(1.9)*sqrt(2.0);
        a += sqrt(1.1)*sqrt(1.2)*sqrt(1.3)*sqrt(1.4)*sqrt(1.5);
        a += sqrt(1.6)*sqrt(1.7)*sqrt(1.8)*sqrt(1.9);
    }
}

void delay(int workload){
    int i;
    int total_workload = workload*DELAY;

    for (i = 0; i < total_workload; i++)
        core_delay();
}

void do_work(int workload){
    int pid = getpid();

    printf("process %d begins\n", pid);
    delay(workload);
    printf("process %d ends\n", pid);
}
```

```c
int main(){
    pid_t procs[20];
    long now = time(NULL);

    for (int j=0; j<20; ++j){
        procs[j] = fork();
        if (procs[j]<0){
            printf("error in fork\n");
            return 0;
        }
        else if (procs[j] == 0){
            struct sched_param param = {.sched_priority=0};
            if (!set_sched_params(now+j+1, now+j+4, (j+2)*K)){
                sched_setscheduler(0, SCHED_HVF, &param);
                do_work(j+1);
            }
            else {
                printf("invalid scheduling params\n");
            }
            exit(0);
        }
    }

    for (int j=0; j<20; ++j){
        waitpid(procs[j], NULL, 0);
    }

    return 0;
}
```

5.3 Test Results

Ένα shell script Που βρίσκεται στον φάκελο process_data_visualization αντιγράφει τα δεδομένα από το kernel debug trace pipe σε ένα αρχείο csv ώστεόλες οι χρήσιμες πληροφορίες να είναι αποθηκευμένες σε αυτό το αρχείο. Το csv αυτό χρησιμοποιείται και στην δημιουργία των γραφημάτων από το Python script.

Το shell script που αντιγράφει τα αποτελέσματα στο csv είναι το παρακάτω:

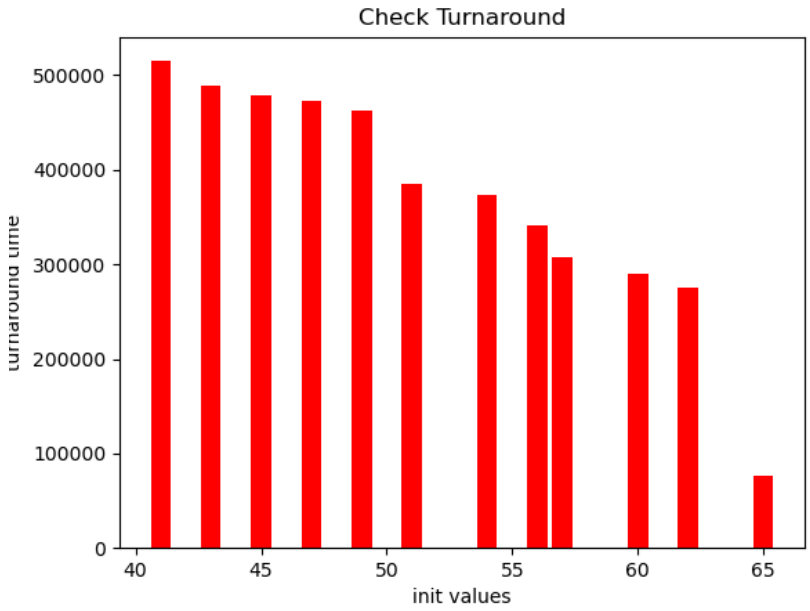
```sh
#!/bin/sh

OUTPUT_FILE="hvf_log.csv"

echo "pid,init_sched_value,turnaround_time,wait_time,computation_time,response_time" > "$OUTPUT_FILE"

cat /sys/kernel/debug/tracing/trace | grep "hvf_task_terminated" | \
    awk -F, '{ print $2","$3","$4","$5","$6","$7 }' >> "$OUTPUT_FILE"
```

τρέχοντας το στην εικονική μηχανή αμέσως μετά από την εκτέλεση του testbench μου δίνεται ως αποτέλεσμα το παρακάτω csv αρχείο.

```csv
pid,init_sched_value,turnaround_time,wait_time,computation_time,response_time
971,65,37734,36058,1676,723
972,65,75957,72711,3246,722
973,62,118934,114048,4886,871
974,62,164857,158222,6635,941
975,62,251880,243604,8276,1007
976,62,276093,266109,9984,1070
977,60,290312,278699,11613,1270
978,57,306892,293994,12898,1329
979,56,340617,325566,15051,1577
980,54,373667,356920,16747,1577
981,51,376242,357896,18346,1577
982,51,385564,365443,20121,1611
984,49,403786,380275,23511,1721
983,49,462264,440456,21808,1719
986,45,472293,445533,26760,1890
985,47,472685,447572,25113,1867
987,45,478773,450161,28612,1888
988,43,488390,458132,30258,1941
989,43,488857,456938,31919,1943
990,41,515058,481096,33962,1994
```

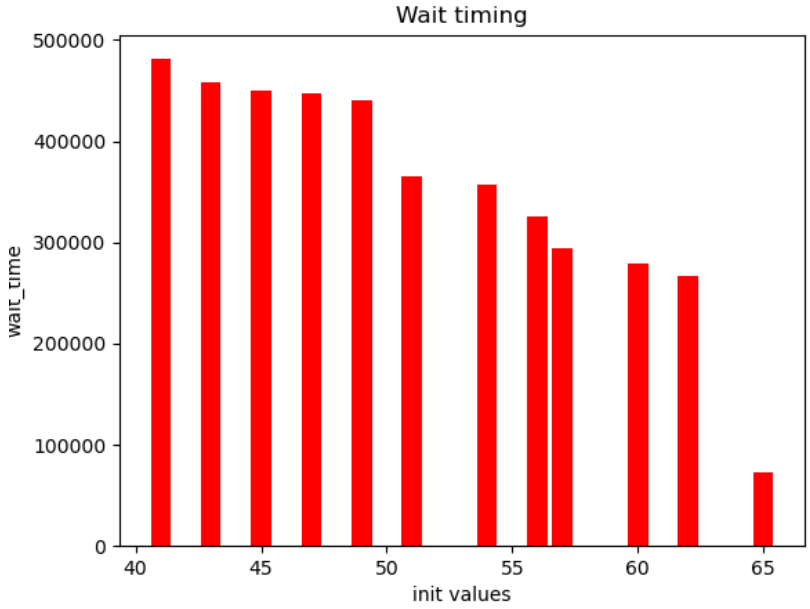
Το python script έπειτα διαβάζει το αρχείο csv που παράχθηκε και παράγει τα παρακάτω γραφήματα:

1. Χρόνος Ολοκλήρωσης συναρτήσει αρχικού σκορ:



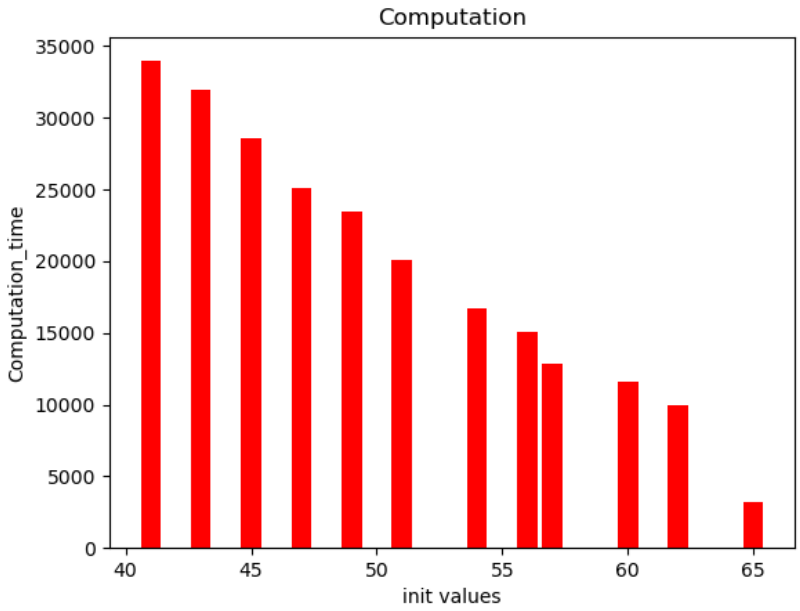
Είναι εύκολο να παρατηρήσουμε ότι όπως ήταν αναμενόμενο οι διεργασίες με το μεγαλύτερο σκορ τείνουν να ολοκληρώνονται σε μικρότερο χρόνο.

2. Χρόνος Αναμονής συναρτήσει αρχικού σκορ



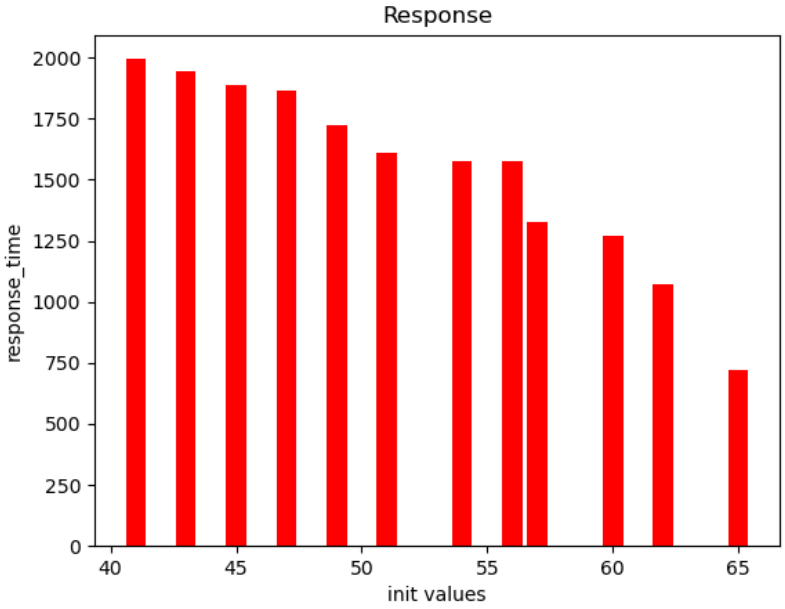
Είναι εύκολο να παρατηρήσουμε ότι όπως ήταν αναμενόμενο οι διεργασίες με το μεγαλύτερο σκορ τείνουν να περιμένουν στην ουρά λιγότερο χρόνο.

3. Χρόνος Εκτέλεσης συναρτήσει αρχικού σκορ



Είναι εύκολο να παρατηρήσουμε ότι όπως ήταν αναμενόμενο οι διεργασίες με το μεγαλύτερο σκορ τείνουν να χρειάζονται μικρότερο χρόνο για να εκτελεστούν.

4. Χρόνος απόδοσης συναρτήσει αρχικού σκορ



Είναι εύκολο να παρατηρήσουμε ότι όπως ήταν αναμενόμενο οι διεργασίες με το μεγαλύτερο σκορτείνουν να λαμβάνουν την πρώτη απάντηση από την cpu σε μικρότερο χρόνο.

Ο κώδικας του Python script που παρήγαγε τα γραφήματα βρίσκεται παρακάτω:

```python
import pandas as pd
import  sys
import mplcursors
import string
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt

# Προσοχη θελει installation pandas


def readcsv(filename):
    df=pd.read_csv(filename)

    tuples=df.apply(lambda row: tuple(row), axis=1)
    return tuples

def creatediagram(x_axis,y_axis,PID,title,x_label,y_label):
    bars = plt.bar(x_axis, y_axis, color='red')




    cursor = mplcursors.cursor(hover=mplcursors.HoverMode.Transient)

    @cursor.connect("add")

    def on_add(sel):
            x, y, width, height = sel.artist[sel.index].get_bbox().bounds
            sel.annotation.set(text="PID: "+ str(PID[sel.index]),
                               position=(0, 20), anncoords="offset points")
            sel.annotation.xy = sel.artist[sel.index].get_xy()
            sel.annotation.xy = (
                sel.artist[sel.index].get_x() + sel.artist[sel.index].get_width() / 2,
                sel.artist[sel.index].get_height()
            )




    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.title(title)
    plt.show()
```

```python
filename=sys.argv[1]

read=readcsv(filename)
processID=[]
init_values=[] #always in x axis

#always in y axis
turnaround_time=[]
wait_time=[]
computation_time=[]
response_time=[]

for i in range (len(read)):
    processID.append(read[i][0])
    init_values.append(float(read[i][1]))
    turnaround_time.append(float(read[i][2]))
    wait_time.append(float(read[i][3]))
    computation_time.append(float(read[i][4]))
    response_time.append(float(read[i][5]))



#create diagrams

creatediagram(init_values,turnaround_time,processID,"Check Turnaround","init values","turnaround time")

creatediagram(init_values,wait_time,processID,"Wait timing","init values","wait_time")

creatediagram(init_values,computation_time,processID,"Computation","init values","Computation_time")

creatediagram(init_values,response_time,processID,"Response","init values","response_time")
```

μέσω αυτού του script διαβάζεται το hvf_log.csv και παραάγονται τα αντίστοιχα γραφήματα που παρουσιάζονται παραπάνω.