$2^{η}$ Εργαστηριακή Άσκηση στο Μάθημα: ‘Λειτουργικά Συστήματα’

Ομάδα Εργασίας: Καλημέρη Σταματία
Λαζανάς Αλέξιος

Τμήμα Μηχανικών Η/Υ και Πληροφορικής (Πανεπιστήμιο Πατρών) 3 Μαρτίου 2025

Περιεχόμενα

0 Γενικά Σχόλια και Παραδοχές

- Για την υλοποίηση της εργασίας χρησιμοποιήθηκε η γλώσσα $C++20$ και πέρα από την βασική βιβλιοθήκη της χρησιμοποιήθηκαν και εργαλεία από την boost-library.
- Θεωρήσαμε ότι το σύστημα έχει καθορισμένο πλήθος επεξεργαστών σταθερό στον αριθμό 4.
- Κάθε επεξεργαστής είναι στην πραγματικότητα μια διεργασία που δημιουργείται με $fork()$ και εκτελεί την χρονοδρομολόγηση με την κατάλληλη πολιτική σε κάθε περίπτωση. Σε μια δεδομένη εκτέλεση όλοι οι επεξεργαστές εκτελούν την ίδια πολιτική.
- Αφού ένας επεξεργαστής έχει εκτελέσει την δρομολόγηση τερματίζεται (με την χρήση της $exit(0)$)
- Η $main$ διεργασία έχει ως διεργασίες-παιδιά τους επεξεργαστές και τους περιμένει για να τελειώσει.
- Η ουρά των διεργασιών προς εκτέλεση βρίσκεται σε κοινή περιοχή μνήμης μαζί με ένα $mutex$ που έχει σκοπό τον έλεγχο της πρόσβασης των διεργασιών σε αυτήν.

1 Φάση $1^{η}$: Χρονοδρομολογητής για σύστημα πολλαπλών επεξεργαστών όπου οι διεργασίες απαιτούν μόνο έναν επεξεργαστή η καθεμιά.

1.1 Γενική δομή διεργασιών και εργαλεία που χρησιμοποιήθηκαν στην υλοποίηση

Οι διεργασίες έχουν δομή παρόμοια με αυτή του κώδικα σε C που μας δόθηκε. Συνεπώς ορίσαμε τύπους δεδομένων:

- Για την κατάσταση της διεργασίας **enum** *proc_status*
- Για την πολιτική χρονοδρομολόγησης **enum** *policy*
- Για την διεργασία **struct** *process*

Οι παραπάνω τύποι δεδομένων είναι ορισμένοι στο αρχείο *scheduler_utils.cpp* (το οποίο αρχείο περιέχει και τους ορισμούς των συναρτήσεων που υλοποιούν τους ζητούμενους αλγορίθμους) και το μέρος του κώδικα που τους ορίζει φαίνεται παρακάτω:

```cpp
#include <string>
#include <semaphore>
#include <unistd.h>
#include <sys/wait.h>
#include <ctime>
#include <chrono>
#include <csignal>
#include <errno.h>
#include <list>
#include <boost/interprocess/shared_memory_object.hpp>
#include <boost/interprocess/mapped_region.hpp>
#include <boost/interprocess/sync/interprocess_mutex.hpp>
#include <boost/interprocess/containers/list.hpp>
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/allocators/allocator.hpp>

#include <sys/wait.h>
#define SHARED_SIZE 65536
using namespace std;
using namespace std::chrono;
namespace bip = boost::interprocess;
namespace bc = boost::container;

typedef struct sigaction sig_action;

time_point<system_clock> global_t;
size_t time_slice=4;
counting_semaphore<10> proc_wake(0);
const size_t num_processors=4;
size_t size_q;



enum proc_status{
    PROC_NEW,
    PROC_STOPPED,
    PROC_RUNNING,
    PROC_EXITED
};


enum policy{
    RR,
    FCFS,
    RRAFF
};


struct process{
    string name;
    size_t pid;
    proc_status status;
    time_point<system_clock> t_submission, t_start, t_end;
    process(){}
    process(string name_, size_t pid_, time_point<system_clock> t_submission_): name(name_), pid(pid_), t_submission(t_submission_), status(PROC_NEW){}
};
```

Η κατάσταση μιας διεργασίας μπορεί να είναι:

1. Νέα διεργασία: ($PROC\_NEW$)
2. Διεργασία που μόλις σταμάτησε ($PROC\_STOPPED$)
3. Διεργασία που εκτελείται αυτή την στιγμή ($PROC\_RUNNING$)
4. Διεργασία ολοκληρωμένη ($PROC\_EXITED$)

Η πολιτική μπορεί να είναι μια από τις τρεις ζητούμενες της πρώτης φάσης (RR, $FCFS$, $RRAFF$)

Μια διεργασία αποτελείται από όνομα($name$), pid, κατάσταση, χρονική στιγμή άφιξης($t\_submission$), χρονική στιγμή έναρξης (t_start) και χρονική στιγμή ολοκλήρωσης (t_end).

Επειδή ο κάθε επεξεργαστής είναι στην ουσία μια διεργασία (διεργασία-γονέας) που δημιουργεί την διεργασία που εκτελείται πάνω σε αυτόν (διεργασία-παιδί) έπρεπε όλοι οι επεξεργαστές να έχουν πρόσβαση σε μια κοινή περιοχή μνήμης, συνεπώς απαιτείτο να ορίσουμε σε ένα **struct** τι θα περιλαμβάνει αυτή η κοινή περιοχή μνήμης. Όπως αναφέρθηκε στα γενικά σχόλια χρειάζεται μια λίστα με διεργασίες και επιπλέον ένα $mutex$ για να διασφαλίζει ατομική πρόσβαση στην λίστα αυτή.

```
struct shared_data{
    using shared_allocator_t = bip::allocator<process, bip::managed_shared_memory::segment_manager>;
    using shared_list_t = bc::list<process, shared_allocator_t>;
    shared_list_t proc_queue;
    bip::interprocess_mutex mtx;

    shared_data(const shared_allocator_t& alloc) : proc_queue(alloc) {}

};
```

επειδή θέλουμε η λίστα να αποθηκεύει τα δεδομένα της στην κοινή μνήμη δεν είναι δυνατή η χρήση της $std :: list$ για την λίστα. Αντίθετα χρειαζόμαστε έναν $allocator$ ο οποίος στην ουσία θα υποδεικνύει στην λίστα σε ποια περιοχή μνήμης θα αποθηκευθούν τα στοιχεία της, για αυτό τον σκοπό χρησιμοποιήσαμε τον $boost :: interprocess :: allocator$ που δεσμεύει μνήμη για αντικείμενα τύπου $process$ σε μια κοινή περιοχή. Λόγω της χρήσης αυτού του $allocator$ έπρεπε να χρησιμοποιήσουμε την λίστα $boost :: container :: list$ η οποία μπορεί να χειριστεί τέτοιους $allocators$.

Για τους προεκτοπιστικούς αλγορίθμους (RR και $RRAFF$) απαιτείται χειρισμός σημάτων. Συγκεκριμένα όταν η διεργασία-παιδί τελειώσει πρέπει να σταλθεί σήμα στην διεργασία-γονέα. Το σήμα που στέλνεται αυτόματα σε αυτή την περίπτωση είναι το $SIGCHLD$ όμως το συγκεκριμένο σήμα στέλνεται στην διαδικασία-γονέα σε κάθε περίπτωση όπου η διεργασία παιδί διακόπτεται (είτε επειδή τερμάτισε, είτε επειδή της ήρθε σήμα $SIGSTOP$, $SIGINT$, $SIGKILL$,...), εμείς επιθυμούσαμε να χειριστούμε καταλλήλως τον τερματισμό της διεργασίας. Συνεπώς κάθε φορά που δημιουργούμε ένα αντικείμενο τύπου **struct** sigaction sa θέτουμε το μέλος sa.sa_flags=SA_SIGINFO ώστε να αναγνωριστούν οι πληροφορίες της λήψης του σήματος από την συνάρτηση χειρισμού του. συνεπώς περίπου στην αρχή κάθε συνάρτησης δρομολόγησης υπάρχει ο εξής κώδικας:

```
sig_action sa;
sa.sa_handler=nullptr;
sa.sa_sigaction=signal_handler;
sa.sa_flags=SA_SIGINFO;
sigemptyset(&sa.sa_mask);
if (sigaction(SIGCHLD, &sa, nullptr) == -1) {
    cerr << "Failed to set up signal handler.\n";
    return;
}
```

έτσι θέτουμε ως κατάλληλο χειριστή του σήματος $SIGCHLD$ την συνάρτηση signal_handler, η οποία λόγω της τιμής του sa_flags πληροφορείται για την κατάσταση όπου το σήμα SIGCHLD ελήφθη από την γονεϊκή διεργασία (δηλ. τον δρομολογητή).

Στην παρακάτω εικόνα υπάρχει ο κώδικας της συνάρτησης-χειριστή του σήματος.

```
volatile sig_atomic_t proc_finished=0;
policy pol;
process* running;

void termination_achieved(){
    running->status=PROC_EXITED;
    running->t_end=system_clock::now();
    cout<<"PID "<<running->pid<< " CMD: "<<running->name<<"\n";
    cout<<"\tElapsed time = "<<duration_cast<seconds>(running->t_end-running->t_submission).count()<< "secs\n";
    cout<<"\tExecution time = "<<duration_cast<seconds>(running->t_end-running->t_start).count()<<"secs\n";
    cout<<"\tWorkload time = "<<duration_cast<seconds>(running->t_end-global_t).count()<<"secs\n";
}

void signal_handler(int signo, siginfo_t* info, void* context){
    if(pol!=FCFS && info->si_code==CLD_EXITED){
        proc_finished=1;
        proc_wake.release();
    }
}
```

Αν μια διεργασία τελειώσει και στείλει στον γονέα της το σήμα $SIGCHLD$ και οι πρόσθετες πληροφορίες θα ενημερώσουν την συνάρτηση χειρισμού για τον τερματισμό, η τρέχουσα θυγατρική διεργασία θα αλλάξει κατάσταση σε ολοκληρωμένη θα αποθηκεύσει την στιγμή τερματισμού της και θα εκτυπώσει τα απαραίτητα δεδομένα. Τέλος θα απελευθερώσει έναν τοπικό σημαφόρο του οποίου την χρήση θα εξηγήσουμε στην υποενότητα που αφορά τις παραλλαγές του αλγορίθμου Round-Robin.

Η συνάρτηση termination_achieved καλείται και όταν αποτυγχάνει οποιαδήποτε από τις δύο κλήσεις της kill δηλαδή είτε η: kill(child_pid, SIGCONT) είτε η: kill(child_pid, SIGSTOP) στους προεκτοπιστικούς αλγορίθμους. Αυτό επιλέξαμε να το κάνουμε επειδή η συνάρτηση kill αποτυγχάνει (δηλαδή επιστρέφει -1) στις εξής τρεις περιπτώσεις:

1. Ο αριθμός του εκάστοτε σήματος δεν αντιπροσωπεύει κάποιο έγκυρο σήμα τότε η μεταβλητή errno=EINVAL, κάτι που δεν μπορεί να συμβεί στο πρόγραμμα μας καθώς τα σήματα SIGCONT, SIGSTOP είναι έγκυρα.

2. Η διεργασία που στέλνει το σήμα δεν έχει την δικαιοδοσία να στείλει σήμα στην διεργασία-παραλήπτη τότε η μεταβλητή errno=EPERM, κάτι που στο πρόγραμμα δεν μπορεί να συμβεί καθώς οι κλήσεις της kill γίνονται από μητρική σε θυγατρική διεργασία.

3. Η διεργασία παραλήπτης δεν βρέθηκε τότε η μεταβλητή errno=ESRCH, αυτή η περίπτωση μπορεί όντως να συμβεί στο πρόγραμμά μας και είναι η μόνη περίπτωση που χειριζόμαστε (περισσότερα στην υποενότητα που αφορά τις παραλλαγές του αλγορίθμου round-robin).

1.2 Ο Αλγόριθμος χρονοδρομολόγησης First Come First Served($FCFS$).

Ο αλγόριθμος $FCFS$ βασίζεται στην λογική του C κώδικα που μας δόθηκε. Καθώς όμως έχουμε να χειριστούμε πολλαπλούς επεξεργαστές έπρεπε να προσθέσουμε κλειδαριές κατά την διαδικασία εισαγωγής μιας διεργασίας στον καθένα επεξεργαστή ξεχωριστά.

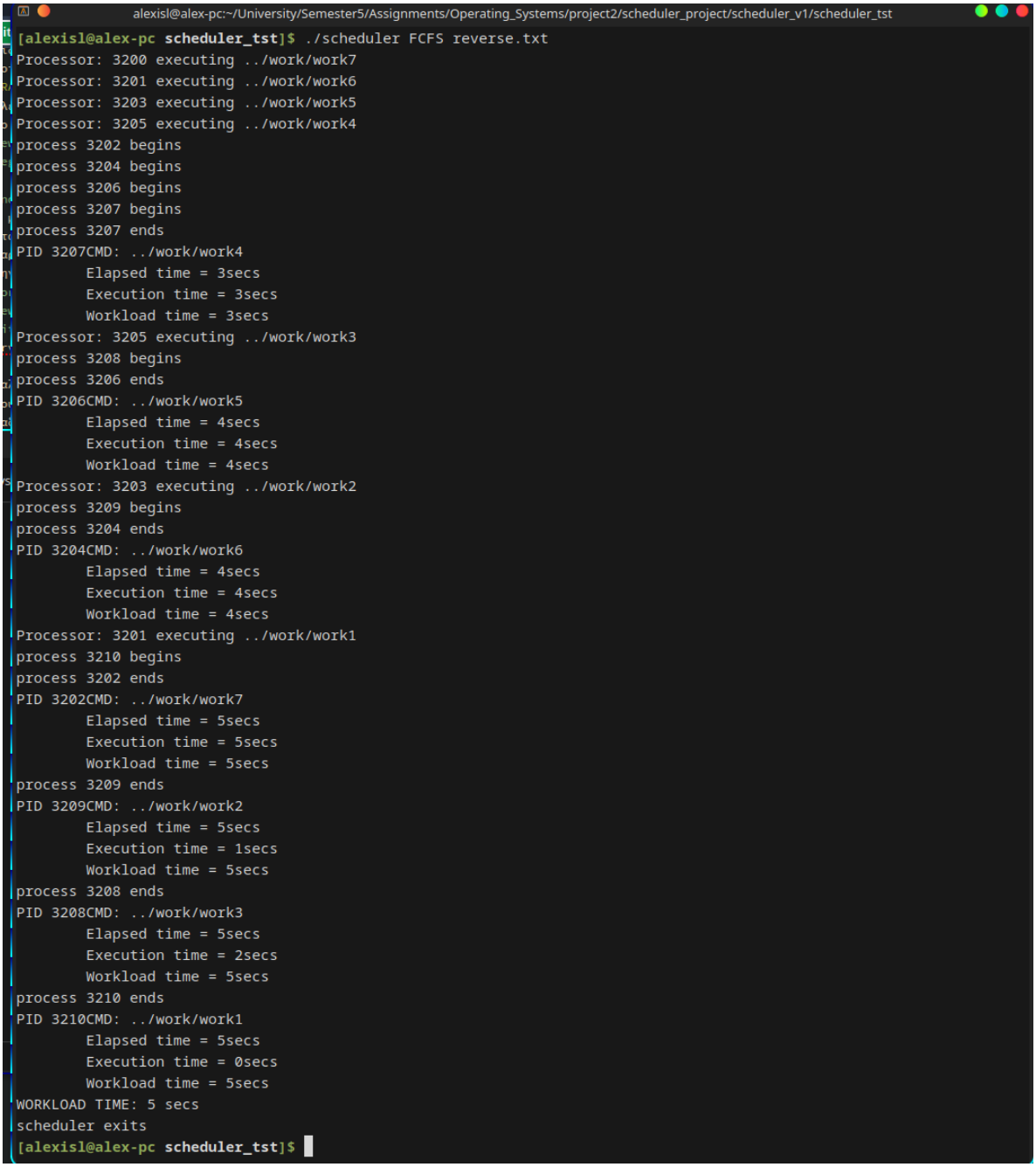
```
void fcfs(shared_data*& shptr){
    process proc;
    int status;
    sig_action sa;
    sa.sa_handler=nullptr;
    sa.sa_sigaction=signal_handler;
    sa.sa_flags=SA_SIGINFO;
    sigemptyset(&sa.sa_mask);
    if (sigaction(SIGCHLD, &sa, nullptr) == -1) {
        cerr << "Failed to set up signal handler.\n";
        return;
    }

    while(true){
        shptr->mtx.lock();
        if(shptr->proc_queue.empty()){
            shptr->mtx.unlock();
            break;
        }
        proc = shptr->proc_queue.front();
        shptr->proc_queue.pop_front();
        shptr->mtx.unlock();
        if(proc.status==PROC_NEW){
            proc.t_start = system_clock::now();
            pid_t pid = fork();
            if(pid<0){
                cerr<<"fork_failed\n";
                exit(-1);
            }
            else if(pid==0){
                cout<<"Processor: "<<getppid()<<" executing "<<proc.name<<"\n";
                execl(proc.name.c_str(), proc.name.c_str(), nullptr);
            }
            else{
                proc.pid=pid;
                proc.status=PROC_RUNNING;
                waitpid(proc.pid, &status, 0);
                proc.status=PROC_EXITED;
                proc.t_end=system_clock::now();
                cout<<"PID "<<pid<< " CMD: "<<proc.name<<"\n";
                cout<<"\tElapsed time = "<<duration_cast<seconds>(proc.t_end-proc.t_submission).count()<< "secs\n";
                cout<<"\tExecution time = "<<duration_cast<seconds>(proc.t_end-proc.t_start).count()<<"secs\n";
                cout<<"\tWorkload time = "<<duration_cast<seconds>(proc.t_end-global_t).count()<<"secs\n";
            }
        }
    }
}
```

Για να γίνουμε πιο συγκεκριμένοι έχουμε βάλει $mutex\_lock$ λίγο πριν ένας επεξεργαστής πάρει μια διεργασία από την ουρά. Στην περίπτωση που η ουρά είναι άδεια έχω $mutex\_unlock$. Σε διαφορετική περίπτωση η διεργασία μπαίνει στον επεξεργαστή απελευθερώνοντας μετέπειτα την κλειδαριά ώστε να μπορούν και οι άλλοι επεξεργαστές να πάρουν τις υπόλοιπες διεργασίες (έτσι διασφαλίζουμε ατομική προσπέλαση στην κοινή ουρά). Όταν ένας επεξεργαστής αφαιρέσει μια διεργασία εκτελείται ο αλγόριθμος $FCFS$ όπως και για στην μονοεπεξεραστική περίπτωση. Δηλαδή η διεργασία που επιλέχθηκε εκτελείται σε κάποιον επεξεργαστή (διεργασία-γονέα). Ο επεξεργαστής απλά περιμένει να τελειώσει η διεργασία που μόλις δρομολογήθηκε ενημερώνει τον χρήστη και προχωρά στην εκτέλεση της επόμενης διαθέσιμης διεργασίας στην ουρά.

Παράδειγμα εκτέλεσης κώδικα για $FCFS$

Για το αρχείο $reverse.txt$:



Για το αρχείο *homogeneous.txt*

```
[alexisl@alex-pc scheduler_tst]$ ./scheduler FCFS homogeneous.txt
Processor: 3610 executing ../work/work7
Processor: 3611 executing ../work/work7
Processor: 3613 executing ../work/work7
Processor: 3615 executing ../work/work7
process 3612 begins
process 3614 begins
process 3616 begins
process 3617 begins
process 3614 ends
PID 3614CMD: ../work/work7
        Elapsed time = 5secs
        Execution time = 5secs
        Workload time = 5secs
Processor: 3611 executing ../work/work7
process 3617 ends
PID 3617CMD: ../work/work7
        Elapsed time = 5secs
        Execution time = 5secs
        Workload time = 5secs
process 3616 ends
process 3618 begins
PID 3616CMD: ../work/work7
        Elapsed time = 5secs
        Execution time = 5secs
        Workload time = 5secs
process 3612 ends
PID 3612CMD: ../work/work7
        Elapsed time = 5secs
        Execution time = 5secs
        Workload time = 5secs
process 3618 ends
PID 3618CMD: ../work/work7
        Elapsed time = 11secs
        Execution time = 5secs
        Workload time = 11secs
WORKLOAD TIME: 11 secs
scheduler exits
[alexisl@alex-pc scheduler_tst]$
```

1.3 Ο Αλγόριθμος χρονοδρομολόγησης Round-Robin(RR)

1.3.1 Όλοι οι επεξεργαστές χειρίζονται μια κοινή ουρά διεργασιών (απλός RR)

Σε αυτή την προσέγγιση, ο κάθε επεξεργαστής δρομολογεί την διεργασία που θα εντοπίσει πρώτη στην ουρά διεργασιών για ένα χρονομερίδιο (του οποίου η διάρκεια εξαρτάται από την είσοδο χρήστη). Αν η διεργασία ολοκληρωθεί σε χρονικό διάστημα μικρότερο του χρονομεριδίου στέλνεται το σήμα $SIGCHLD$ το οποίο χειρίζεται η συνάρτηση $signal\_handler$, η οποία όπως γράφτηκε στην εισαγωγή στο τέλος απελευθερώνει έναν σημαφόρο ο οποίος χρησιμοποιείται για αυτόν ακριβώς τον σκοπό όπως φαίνεται στον παρακάτω κώδικα:

```
void rr(shared_data*& shptr){
    sig_action sa;
    sa.sa_handler=nullptr;
    sa.sa_sigaction=signal_handler;
    sa.sa_flags=SA_SIGINFO;
    sigemptyset(&sa.sa_mask);
    if (sigaction(SIGCHLD, &sa, nullptr) == -1) {
        cerr << "Failed to set up signal handler.\n";
        return;
    }

    while(true){
        shptr->mtx.lock();
        if(shptr->proc_queue.empty()){
            shptr->mtx.unlock();
            break;
        }
        auto proc=shptr->proc_queue.front();
        shptr->proc_queue.pop_front();
        shptr->mtx.unlock();
        if(proc.status==PROC_NEW){
            proc.t_start=system_clock::now();
            proc_finished=0;
            pid_t pid=fork();
            if(pid==0){
                cout<<"Processor: "<<getppid()<<" executing "<<proc.name<<"\n";
                execl(proc.name.c_str(), proc.name.c_str(), nullptr);
            }
            else{
                proc.pid=pid;
                proc.status=PROC_RUNNING;
                running=&proc;
                proc_wake.try_acquire_for(seconds(time_slice));
                if(proc_finished==0){
                    int try_kill=kill(pid, SIGSTOP);
                    if(try_kill==0){
                        proc.status=PROC_STOPPED;
                        cout<<"Processor: "<<getpid()<<" stopped: "<<proc.name<<"\n";
                        shptr->mtx.lock();
                        shptr->proc_queue.push_back(proc);
                        shptr->mtx.unlock();
                    }
                    else if(try_kill==-1 && errno==ESRCH){
                        termination_achieved();
                    }
                }
                else{
                    termination_achieved();
                }
            }
        }
        else if(proc.status==PROC_STOPPED){
            proc_finished=0;
            int try_kill=kill(proc.pid, SIGCONT);
            if(try_kill==0){
                cout<<"Processor: "<<getpid()<<" resumed: "<<proc.name<<"\n";
                proc.status=PROC_RUNNING;
                running=&proc;
                proc_wake.try_acquire_for(seconds(time_slice));
                if(proc_finished==0){
                    int try_kill=kill(proc.pid, SIGSTOP);
                    if(try_kill==0){
                        proc.status=PROC_STOPPED;
                        cout<<"Processor: "<<getpid()<<" stopped: "<<proc.name<<"\n";
                        shptr->mtx.lock();
                        shptr->proc_queue.push_back(proc);
                        shptr->mtx.unlock();
                    }
                    else if(try_kill==-1 && errno==ESRCH){
                        termination_achieved();
                    }
                }
                else{
                    termination_achieved();
                }
            }
            else if(try_kill==-1 && errno==ESRCH){
                termination_achieved();
            }
        }
    }
}
```

Με την εντολή $std :: counting\_semaphore :: try\_acquire\_for(< duration >)$ ο επεξεργαστής μόλις

ξεκινήσει μια διεργασία (είτε με $fork()$ αν είναι νέα διεργασία είτε με $SIGCONT$ αν είναι διεργασία σταματημένη) περιμένει μέχρις ώτου:

1. Το χρονομερίδιο να ξεπεραστεί
2. Ο σημαφόρος να απελευθερωθεί

Ο σημαφόρος απελευθερώνεται όταν ολοκληρωθεί ο χειρισμός του σήματος $SIGCHLD$ από τον $signal\_handler$, αν αυτό εκτελεστεί σε διάρκεια μικρότερη από ένα χρονομερίδιο η κατάσταση της διεργασίας θα μεταβληθεί από τρέχουσα ($PROC\_RUNNING$) σε ολοκληρωμένη ($PROC\_EXITED$) συνεπώς δεν θα εκτελεσθούν οι εντολές στο σώμα της **if** και ο δρομολογητής θα περάσει στην επόμενη διεργασία.

Αντίθετα αν ο σημαφόρος απελευθερωθεί λόγω λήξης του χρονομεριδίου και το σήμα $SIGCHLD$ δεν έχει σταλεί από την διεργασία-παιδί, η κατάσταση της διεργασίας-παιδί είναι ακόμα $PROC\_RUNNING$, οι εντολές στο σώμα της **if** θα εκτελεσθούν, η διεργασία θα σταματήσει (θα της σταλεί $SIGSTOP$) και θα επανεισαχθεί με ασφαλή και ατομικό μηχανισμό (δηλ. $mutex\_lock$ και $mutex\_unlock$) στην κοινή ουρά διεργασιών με κατάσταση $PROC\_STOPPED$ και θα περιμένει εκεί μέχρις ώτου κάποιος επεξεργαστής να την αφαιρέσει από την ουρά (επειδή την βρήκε στην κορυφή της) και να την ξαναξεκινήσει στέλνοντάς της σήμα $SIGCONT$.

Οποιαδήποτε αποτυχία της $kill$ επειδή η διεργασία δεν βρέθηκε δείχνει ότι η διεργασία έχει τελειώσει νωρίτερα αλλά λόγω κάποιου race-condition το σήμα SIGCHLD δεν χειρίσθηκε σωστά, συνεπώς αν συμβεί αυτό καλείται η συνάρτηση termination-achieved η οποία φροντίζει να μην ξαναδρομολογηθεί η συγκεκριμένη διεργασία καθώς είναι ολοκληρωμένη.

Παραδείγματα εκτέλεσης του RR (Το χρονομερίδιο εισάγεται σαν είσοδο από τον χρήστη σε ms)

Για το αρχείο $homogeneous.txt$

```
[alexisl@alex-pc scheduler]$ ./scheduler RR 2000 homogeneous.txt
Processor: 11700 executing ../work/work7
Processor: 11701 executing ../work/work7
Processor: 11703 executing ../work/work7
Processor: 11705 executing ../work/work7
process 11702 begins
process 11704 begins
process 11706 begins
process 11707 begins
Processor: 11700 stopped: ../work/work7
Processor: 11701 stopped: ../work/work7
Processor: 11701 resumed: ../work/work7
Processor: 11703 stopped: ../work/work7
Processor: 11703 resumed: ../work/work7
Processor: 11705 stopped: ../work/work7
Processor: 11705 resumed: ../work/work7
Processor: 11700 executing ../work/work7
process 11708 begins
Processor: 11700 stopped: ../work/work7
Processor: 11705 stopped: ../work/work7
Processor: 11700 resumed: ../work/work7
Processor: 11705 resumed: ../work/work7
Processor: 11701 stopped: ../work/work7
Processor: 11701 resumed: ../work/work7
Processor: 11703 stopped: ../work/work7
Processor: 11703 resumed: ../work/work7
process 11706 ends
PID 11702 CMD: ../work/work7
        Elapsed time = 5secs
        Execution time = 5secs
        Workload time = 5secs
Processor: 11703 resumed: ../work/work7
process 11702 ends
PID 11707 CMD: ../work/work7
        Elapsed time = 5secs
        Execution time = 5secs
        Workload time = 5secs
Processor: 11705 stopped: ../work/work7
Processor: 11705 resumed: ../work/work7
Processor: 11701 stopped: ../work/work7
Processor: 11701 resumed: ../work/work7
process 11704 ends
PID 11706 CMD: ../work/work7
        Elapsed time = 7secs
        Execution time = 7secs
        Workload time = 7secs
process 11708 ends
process 11707 ends
PID 11708 CMD: ../work/work7
        Elapsed time = 7secs
        Execution time = 5secs
        Workload time = 7secs
PID 11704 CMD: ../work/work7
        Elapsed time = 7secs
        Execution time = 7secs
        Workload time = 7secs
WORKLOAD TIME: 7 secs
scheduler exits
[alexisl@alex-pc scheduler]$
```

Για το αρχείο *reverse.txt*

```
[alexisl@alex-pc scheduler]$ ./scheduler RR 2000 reverse.txt
Processor: 11042 executing ../work/work7
Processor: 11043 executing ../work/work6
Processor: 11045 executing ../work/work5
Processor: 11047 executing ../work/work4
process 11044 begins
process 11046 begins
process 11049 begins
process 11048 begins
Processor: 11042 stopped: ../work/work7
Processor: 11043 stopped: ../work/work6
Processor: 11045 stopped: ../work/work5
Processor: 11047 stopped: ../work/work4
Processor: 11043 executing ../work/work2
Processor: 11047 resumed: ../work/work7
Processor: 11042 executing ../work/work3
Processor: 11045 executing ../work/work1
process 11050 begins
process 11052 begins
process 11051 begins
process 11052 ends
PID 11052 CMD: ../work/work1
        Elapsed time = 2secs
        Execution time = 0secs
        Workload time = 2secs
Processor: 11045 resumed: ../work/work6
process 11050 ends
PID 11050 CMD: ../work/work2
        Elapsed time = 3secs
        Execution time = 1secs
        Workload time = 3secs
Processor: 11043 resumed: ../work/work5
Processor: 11047 stopped: ../work/work7
Processor: 11047 resumed: ../work/work4
Processor: 11042 stopped: ../work/work3
Processor: 11042 resumed: ../work/work7
Processor: 11045 stopped: ../work/work6
Processor: 11045 resumed: ../work/work3
process 11049 ends
PID 11049 CMD: ../work/work4
        Elapsed time = 5secs
        Execution time = 5secs
        Workload time = 5secs
Processor: 11047 resumed: ../work/work6
process 11051 ends
PID 11044 CMD: ../work/work7
        Elapsed time = 5secs
        Execution time = 5secs
        Workload time = 5secs
Processor: 11043 stopped: ../work/work5
Processor: 11043 resumed: ../work/work5
process 11048 ends
PID 11051 CMD: ../work/work3
        Elapsed time = 5secs
        Execution time = 3secs
        Workload time = 5secs
process 11044 ends
process 11046 ends
PID 11048 CMD: ../work/work5
        Elapsed time = 6secs
        Execution time = 6secs
        Workload time = 6secs
PID 11046 CMD: ../work/work6
        Elapsed time = 7secs
        Execution time = 7secs
        Workload time = 7secs
WORKLOAD TIME: 7 secs
scheduler exits
[alexisl@alex-pc scheduler]$
```

Όπως φαίνεται και από την έξοδο, μια διεργασία σταματά από τον επεξεργαστή όπου την ξεκίνησε πιο πρόσφατα και μπορεί να την συνεχίσει, δηλαδή να της στείλει σήμα $SIGCONT$ κάποιος άλλος. Όπως στην έξοδο του παραδείγματος, η $work7$ ξεκινά στον επεξεργαστή 11042 οποίος την σταματά στην λήξη του χρονομεριδίου, και ο επεξεργαστής που την συνεχίζει αργότερα είναι ο 11047 ο οποίος την σταματά επίσης για να την υποδεχτεί έπειτα ξανά ο 11042 στον οποίο και ολοκληρώνεται.

Ο συνολικός χρόνος εκτέλεσης είναι πιθανό να διαφέρει σε διαφορετικές συνθήκες εκτέλεσης των προγραμμάτων.

1.3.2 Κάθε επεξεργαστής εκτελεί διεργασίες από μια δική του ουρά διεργασιών($RRAFF$)

Ο αλγόριθμος ακολουθεί την λογική του RR με μόνη διαφορά ότι ο επεξεργαστής που πήρε την διεργασία εξαρχής ο ίδιος πρέπει να την ξαναπάρει. Αυτό το έχουμε επιτύχει κάνοντας τα εξής:

1. Κάθε ένας επεξεργαστής έχει από μια λίστα διεργασιών

2. Κάθε επεξεργαστής παίρνει έναν συγκεκριμένο αριθμό διεργασιών το οποίο καθορίζεται από την αρχή (πριν την εκτέλεση του αλγορίθμου). Αυτός ο αριθμός καθορίζεται από το πλήθος των διεργασιών δια το πλήθος των επεξεργαστών συν 1.

3. Επειδή έχουμε πολλούς επεξεργαστές επιβάλλεται η χρήση $mutex$.Συγκεκριμένα, υπάρχουν κατά την διάρκεια που ένας επεξεργαστής παίρνει ένα πλήθος διεργασιών. Όταν τελειώσει απελευθερώνει το $mutex$ επιτρέποντας και στους άλλους να γεμίσουν την ουρά τους με διεργασίες.

4. Πριν μπει η κάθε διεργασία ξεχωριστά στην ουρά κάθε επεξεργαστή ελέγχεται πάντα αν η λίστα διεργασιών είναι άδεια. Στην περίπτωση που είναι άδεια η λίστα αλλά δεν έχει γεμίσει η ουρά του επεξεργαστή, ο επεξεργαστής συνεχίζει με όσες έχει πάρει

5. Αφού απελευθερώσει το $mutex$ ένας επεξεργαστής εκτελεί ανεξάρτητα από τους άλλους τον αλγόριθμο RR στην δική του ουρά.

Κώδικας:

```
void rraff(shared_data*& shptr){
    sig_action sa;
    sa.sa_handler=nullptr;
    sa.sa_sigaction=signal_handler;
    sa.sa_flags=SA_SIGINFO;
    sigemptyset(&sa.sa_mask);
    if (sigaction(SIGCHLD, &sa, nullptr) == -1) {
        cerr << "Failed to set up signal handler.\n";
        return;
    }

    process proc;
    for(int i=1;i<size_q;i++){
        shptr->mtx.lock();
        if(shptr->proc_queue.empty() && local_q.empty()){
            shptr->mtx.unlock();
            break;
        }
        if(!(shptr->proc_queue.empty())){
            proc = shptr->proc_queue.front();
            shptr->proc_queue.pop_front();
            shptr->mtx.unlock();
            local_q.push_back(proc);
        }
        else{
            shptr->mtx.unlock();
            break;
        }
    }

    while(true){
        if(local_q.empty()){
            break;
        }
        auto tmp=local_q.front();
        local_q.pop_front();

        if(tmp.status==PROC_NEW){
            tmp.t_start=system_clock::now();
            proc_finished=1;
            pid_t pid=fork();
            if(pid==0){
                cout<<"Processor: "<<getppid()<<" executing "<<tmp.name<<"\n";
                execl(tmp.name.c_str(), tmp.name.c_str(), nullptr);
            }
            else{
                tmp.pid=pid;
                tmp.status=PROC_RUNNING;
                running=&tmp;
                proc_wake.try_acquire_for(seconds(time_slice));
                if(proc_finished==1){
                    int try_kill=kill(pid, SIGSTOP);
                    if(try_kill==0){
                        tmp.status=PROC_STOPPED;
                        cout<<"Processor: "<<getpid()<<" stopped: "<<tmp.name<<"\n";
                        local_q.push_back(tmp);
                    }
                    else if(try_kill==-1 && errno==ESRCH){
                        termination_achieved();
                    }
                }
                else{
                    termination_achieved();
                }
            }
        }
        else{
            proc_finished=0;
            int try_kill=kill(tmp.pid, SIGCONT);
            if(try_kill==0){
                cout<<"Processor: "<<getpid()<<" resumed: "<<tmp.name<<"\n";
                tmp.status=PROC_RUNNING;
                running=&tmp;
                proc_wake.try_acquire_for(seconds(time_slice));
                if(proc_finished==0){
                    int try_kill=kill(tmp.pid, SIGSTOP);
                    if(try_kill==0){
                        tmp.status=PROC_STOPPED;
                        cout<<"Processor: "<<getpid()<<" stopped: "<<tmp.name<<"\n";
                        local_q.push_back(tmp);
                    }
                    else if(try_kill==-1 && errno==ESRCH){
                        termination_achieved();
                    }
                }
                else{
                    termination_achieved();
                }
            }
            else if(try_kill==-1 && errno==ESRCH){
                termination_achieved();
            }
        }
    }
}
```

Και σε αυτή την εκδοχή ισχύουν τα ίδια με την προηγούμενη για την περίπτωση αποτυχίας της *kill*

Παράδειγμα εκτέλεσης κώδικα (Το χρονομερίδιο εισάγεται σαν είσοδο από τον χρήστη σε ms):
Για *reverse.txt*:

```
[alexisl@alex-pc scheduler]$ ./scheduler RRAFF 2000 reverse.txt
Processor: 12552 executing ../work/work7
Processor: 12553 executing ../work/work5
Processor: 12554 executing ../work/work3
Processor: 12557 executing ../work/work1
process 12555 begins
process 12556 begins
process 12558 begins
process 12559 begins
process 12559 ends
PID 12559 CMD: ../work/work1
        Elapsed time = 0secs
        Execution time = 0secs
        Workload time = 0secs
Processor: 12552 stopped: ../work/work7
Processor: 12553 stopped: ../work/work5
Processor: 12554 stopped: ../work/work3
Processor: 12552 executing ../work/work6
Processor: 12553 executing ../work/work4
Processor: 12554 executing ../work/work2
process 12561 begins
process 12560 begins
process 12562 begins
process 12562 ends
PID 12562 CMD: ../work/work2
        Elapsed time = 3secs
        Execution time = 1secs
        Workload time = 3secs
Processor: 12554 resumed: ../work/work3
Processor: 12552 stopped: ../work/work6
Processor: 12552 resumed: ../work/work7
Processor: 12553 stopped: ../work/work4
Processor: 12553 resumed: ../work/work5
process 12558 ends
PID 12558 CMD: ../work/work3
        Elapsed time = 4secs
        Execution time = 4secs
        Workload time = 4secs
Processor: 12553 resumed: ../work/work4
Processor: 12552 stopped: ../work/work7
Processor: 12552 resumed: ../work/work6
process 12561 ends
PID 12561 CMD: ../work/work4
        Elapsed time = 7secs
        Execution time = 5secs
        Workload time = 7secs
Processor: 12553 resumed: ../work/work5
process 12556 ends
PID 12556 CMD: ../work/work5
        Elapsed time = 7secs
        Execution time = 7secs
        Workload time = 7secs
Processor: 12552 stopped: ../work/work6
Processor: 12552 resumed: ../work/work7
process 12555 ends
PID 12555 CMD: ../work/work7
        Elapsed time = 9secs
        Execution time = 9secs
        Workload time = 9secs
Processor: 12552 resumed: ../work/work6
process 12560 ends
PID 12560 CMD: ../work/work6
        Elapsed time = 10secs
        Execution time = 8secs
        Workload time = 10secs
WORKLOAD TIME: 10 secs
scheduler exits
[alexisl@alex-pc scheduler]$
```

Για *homogeneous.txt*:

```
[alexisl@alex-pc scheduler]$ ./scheduler RRAFF 2000 homogeneous.txt
Processor: 12214 executing ../work/work7
Processor: 12215 executing ../work/work7
Processor: 12217 executing ../work/work7
process 12219 begins
process 12220 begins
process 12216 begins
Processor: 12214 stopped: ../work/work7
Processor: 12215 stopped: ../work/work7
Processor: 12217 stopped: ../work/work7
Processor: 12217 resumed: ../work/work7
Processor: 12215 executing ../work/work7
Processor: 12214 executing ../work/work7
process 12223 begins
process 12224 begins
Processor: 12217 stopped: ../work/work7
Processor: 12215 stopped: ../work/work7
Processor: 12217 resumed: ../work/work7
Processor: 12215 resumed: ../work/work7
Processor: 12214 stopped: ../work/work7
Processor: 12214 resumed: ../work/work7
process 12220 ends
PID 12220 CMD: ../work/work7
        Elapsed time = 5secs
        Execution time = 5secs
        Workload time = 5secs
Processor: 12214 stopped: ../work/work7
Processor: 12214 resumed: ../work/work7
Processor: 12215 stopped: ../work/work7
Processor: 12215 resumed: ../work/work7
Processor: 12214 stopped: ../work/work7
Processor: 12214 resumed: ../work/work7
Processor: 12215 stopped: ../work/work7
Processor: 12215 resumed: ../work/work7
process 12219 ends
PID 12219 CMD: ../work/work7
        Elapsed time = 9secs
        Execution time = 9secs
        Workload time = 9secs
Processor: 12215 resumed: ../work/work7
process 12216 ends
PID 12216 CMD: ../work/work7
        Elapsed time = 9secs
        Execution time = 9secs
        Workload time = 9secs
Processor: 12214 resumed: ../work/work7
process 12223 ends
PID 12223 CMD: ../work/work7
        Elapsed time = 11secs
        Execution time = 9secs
        Workload time = 11secs
process 12224 ends
PID 12224 CMD: ../work/work7
        Elapsed time = 11secs
        Execution time = 9secs
        Workload time = 11secs
WORKLOAD TIME: 11 secs
scheduler exits
[alexisl@alex-pc scheduler]$
```

Ο συνολικός χρόνος εκτέλεσης είναι πιθανό να διαφέρει σε διαφορετικές συνθήκες εκτέλεσης των προγραμμάτων. Σχόλια περί εκτέλεσης:

Στην περίπτωση που κάποιος επεξεργαστής δεν καταφέρει να ολοκληρώσει κάποια διεργασία σε ένα κβάντο, όπως με τον 12214 που εκτελεί την διεργασία work7. Παρατηρείται από την έξοδο του προγράμματος πως ο επεξεργαστής που σταμάτησε την διεργασία είναι ο ίδιος με εκείνον που θα την εκκινήσει και πάλι. Πράγματι για τον 12214 έχω τα εξής μηνύματα:

Processor:12214 executing ../work/work7, Processor:12214 stopped: ../work/work7, Processor:12214 resumed: ../work/work7.

Έτσι είναι φανερό ότι η διεργασία work7 είναι συνδεδεμένη μοναδικά με τον επεξεργαστή 12214.

1.4 Η διεπαφή και η λειτουργία του χρονοδρομολογητή

Η διεπαφή του προγράμματος αναδείχτηκε και στα παραδείγματα εκτέλεσης στις δυο προηγούμενες υποενότητες σε αυτή την ενότητα θα επεξηγηθεί το αρχείο *scheduler.cpp* όπου ορίζεται η συνάρτηση *main* που επεξεργάζεται την είσοδο του χρήστη και ανοίγει το ανάλογο αρχείο, δημιουργεί τους επεξεργαστές και ο καθένας από αυτούς εκτελεί την πολιτική που έχει επιλεχθεί. Ο κώδικας φαίνεται παρακάτω:

```
int main(int argc, char* argv[]){
    map<string, policy> pol_map;
    pol_map["FCFS"]=FCFS;
    pol_map["RR"]=RR;
    pol_map["RRAFF"]=RRAFF;

    bip::shared_memory_object::remove("shm");

    bip::managed_shared_memory shm(bip::open_or_create, "shm", SHARED_SIZE);
    shared_data* shptr = shm.find_or_construct<shared_data>("sharedData")(shm.get_segment_manager());
    pol=RR;
    global_t=system_clock::now();
    fstream fin;
    if (argc == 1) {
        cerr<<"invalid usage\n";
        exit(1);
    }
    else if (argc == 2) {
        fin.open(argv[1]);
        if(!fin.is_open()){
            cerr<<"invalid input filename\n";
            exit(1);
        }
    }
    else if (argc > 2) {
        if (!strcmp(argv[1],"FCFS")) {
            pol = FCFS;
            fin.open(argv[2]);
            if(!fin.is_open()){
                cerr<<"invalid input filename\n";
                exit(1);
            }
        }
        else if (!strcmp(argv[1],"RR") || !strcmp(argv[1],"RRAFF")) {
            pol = pol_map[argv[1]];
            time_slice = stoi(argv[2])/1000;
            fin.open(argv[3]);
            if(!fin.is_open()){
                cerr<<"invalid input filename\n";
                exit(1);
            }
        }
        else {
            cerr<<"invalid usage\n";
            exit(1);
        }
    }
    if(time_slice<2){
        cerr<<"too small time slice\n";
        exit(1);
    }

    if(fin.is_open()){
        string tmp_name;
        while(getline(fin, tmp_name)){
            process proc(tmp_name, 0, global_t);
            shptr->proc_queue.push_back(proc);
        }
    }
    fin.close();
    vector<pid_t> processors;
    size_t num_pros=shptr->proc_queue.size();
    size_q=(num_pros/num_processors)+1;

    for(size_t i=0; i<num_processors; ++i){
        pid_t pid=fork();
        if(pid==0){
            choose_policy(shptr);
        }
        else if(pid>0){
            processors.push_back(pid);
        }
    }

    for (pid_t cpu : processors) {
        int status;
        waitpid(cpu, &status, 0);
    }

    bip::shared_memory_object::remove("shm");

    cout<<"WORKLOAD TIME: "<<duration_cast<seconds>(system_clock::now()-global_t).count()<<" secs\n";
    cout<<"scheduler exits\n";
}


void choose_policy(shared_data*& shptr){
    switch(pol){
    case FCFS:
        fcfs(shptr);
        break;
    case RR:
        rr(shptr);
        break;
    case RRAFF:
        rraff(shptr);
        break;
    }
    exit(0);
}
```

Ένας mapper αντιστοιχίζει αλφαριθμητικά στις κατάλληλες πολιτικές δρομολόγησης. Δημιουργείται μια κοινή περιοχή μνήμης που θα περιέχει όλα όσα συζητήθηκαν για αυτήν σε προηγούμενη υποενότητα και έπειτα ελέγχεται η είσοδος του χρήστη αν ακολουθεί το σωστό format κατόπιν διαβάζεται το αρχείο που ο χρήστης εισήγαγε, και έπειτα δημιουργούνται οι επεξεργαστές μέσω της **fork**().

Αν ο χρήστης εισάγει στην διεπαφή μόνο όνομα αρχείου η default πολιτική είναι RR με χρονομερίδιο ίσο με 4 sec.

Επειδή με χρονομερίδιο ίσο με 1 δευτερόλεπτο παρατηρήθηκαν προβλήματα ο χρήστης πρέπει να εισάγει χρονομερίδιο μεγαλύτερο ή ίσο του 2000.

Ο κάθε επεξεργαστής εκτελεί την πολιτική που επιλέχθηκε και έπειτα τερματίζεται μέσω της **exit**(**0**). Η κύρια διεργασία περιμένει να τελειώσουν όλοι οι επεξεργαστές (στον βρόχο που καλεί την **waitpid**()) και έπειτα εμφανίζει τον συνολικό χρόνο εκτέλεσης και τερματίζει και αυτή με την σειρά της.

Έχουμε παραδώσει επιπλέον και ένα shell-script για να μεταφράζει το πρόγραμμά μας το compile.sh το οποίο περιέχει την εντολή που χρησιμοποιείται για την μετάφραση του προγράμματος.

```
#!/bin/sh

g++ -std=c++20 -o scheduler $1 -lboost_system
```

έτσι εκτελώντας στο τερματικό την εντολή ./compile.sh scheduler.cpp εξάγεται ο εκτελέσιμος κώδικας. Χρησιμοποιούμε boost-library και counting_semaphores συνεπώς χρειαζόμαστε $C++20$ και το flag -lboost_system ώστε το πρόγραμμα να μεταφραστεί σωστά.

2 Φάση 2^{η} : Χρονοδρομολογητής για σύστημα πολλαπλών επεξεργαστών όπου οι διεργασίες απαιτούν περισσότερους από έναν επεξεργαστή η καθεμιά.

2.1 Αλλαγές σε γενικά στοιχεία της δομής της πρώτης φάσης

Εφόσον η μόνη πολιτική δρομολόγησης που υλοποιείται είναι ο αλγόριθμος $FCFS$ δεν χρειαζόμαστε πλέον τύπο δεδομένων για αυτήν ούτε τρόπο χειρισμού σημάτων τέλος δεν χρειάζεται πλέον ο προσδιορισμός της πολιτικής δρομολόγησης από τον χρήστη οπότε στην διεπαφή δίνεται μόνο το αρχείο με τις διεργασίες προς δρομολόγηση.

Οι καταστάσεις μιας διεργασίας παραμένουν ίδιες για αυτό δεν αλλάζει η **enum** proc_status. Πλέον όμως κάθε διεργασία πρέπει να περιέχει την πληροφορία για τους απαιτούμενους επεξεργαστές της. Συνεπώς μια ακόμα ακέραια μεταβλητή θα εισαχθεί στο **struct** process και τέλος χρειαζόμαστε μια ακόμα μεταβλητή που δίνει το πλήθος των διαθέσιμων επεξεργαστών, η μεταβλητή αυτή αποθηκεύεται σε κοινή περιοχή μνήμης και αρχικοποιείται σε τιμή ίση με το πλήθος των επεξεργαστών.

```
struct process{
    string name;
    size_t pid;
    proc_status status;
    size_t demands;
    time_point<system_clock> t_submission, t_start, t_end;
    process(){}
    process(string name_, size_t pid_, time_point<system_clock> t_submission_, size_t demands_): name(name_), pid(pid_), t_submission(t_submission_), status(PROC_NEW), demands(demands_){}
};

struct shared_data{
    using shared_allocator_t = bip::allocator<process, bip::managed_shared_memory::segment_manager>;
    using shared_list_t = bc::list<process, shared_allocator_t>;
    shared_list_t proc_queue;
    bip::interprocess_mutex mtx;
    size_t available_processors;
    shared_data(const shared_allocator_t& alloc) : proc_queue(alloc), available_processors(num_processors) {}

};
```

Δεν απαιτείται σε αυτή την εκδοχή χειρισμός σημάτων οπότε ότι αφορούσε αυτό το κομμάτι έχει αφαιρεθεί. Το **txt** αρχείο με τα ονόματα των διεργασιών πρέπει να διευκρινίζει και το πλήθος των επεξεργαστών που κάθε διεργασία απαιτεί, συνεπώς η δομή του είναι πλέον της μορφής:

```
../work/work7,3
../work/work6,1
../work/work5,2
../work/work4,1
../work/work3,3
../work/work2,1
../work/work1,1
```

δεξιά από το κόμμα βρίσκεται το πλήθος επεξεργαστών όπου η κάθε διεργασία απαιτεί.

Ανάλογα το αρχείο scheduler.cpp έχει μεταβληθεί ώστε να ακολουθεί την νέα διεπαφή

```cpp
int main(int argc, char* argv[]){
    bip::shared_memory_object::remove("shm");

    bip::managed_shared_memory shm(bip::open_or_create, "shm", SHARED_SIZE);
    shared_data* shptr = shm.find_or_construct<shared_data>("sharedData")(shm.get_segment_manager());
    global_t=system_clock::now();
    fstream fin;
    if (argc == 2) {
        fin.open(argv[1]);
    }
    else{
        cerr<<"invalid usage\n";
        exit(1);
    }

    if(fin.is_open()){
        string tmp_name;
        string trash;
        size_t tmp_proc_num;
        while(getline(fin, tmp_name, ',')){
            fin>>tmp_proc_num;
            getline(fin, trash);
            process proc(tmp_name, 0, global_t, tmp_proc_num);
            shptr->proc_queue.push_back(proc);
        }
    }
    fin.close();
    vector<pid_t> processors;
    size_t num_pros=shptr->proc_queue.size();
    size_t size_q=(num_pros/num_processors)+1;

    for(size_t i=0; i<num_processors; ++i){
        pid_t pid=fork();
        if(pid==0){
            fcfs(shptr);
        }
        else if(pid>0){
            processors.push_back(pid);
        }
    }

    for (pid_t cpu : processors) {
        int status;
        waitpid(cpu, &status, 0);
    }
    bip::shared_memory_object::remove("shm");

    cout<<"WORKLOAD TIME: "<<duration_cast<seconds>(system_clock::now()-global_t).count()<<" secs\n";
    cout<<"scheduler exits\n";
}
```

πλέον ως όρισμα εισόδου δίνεται μόνο το όνομα του αρχείου το οποίο διαβάζεται με βάσει την νέα του μορφή. Πλέον κάθε επεξεργαστής εκτελεί την συνάρτηση $FCFS$, η οποία στο τέλος της καλεί την **exit(0)** και τερματίζει την δρομολόγηση, η συνάρτηση αυτή περιγράφεται στην επόμενη υποενότητα.

2.2 Αλλαγές στον αλγόριθμο First Come First Served($FCFS$)

Ο αλγόριθμος χρονοδρομολόγησης στην περίπτωση αυτή ελέγχει αν υπάρχουν διαθέσιμοι επεξεργαστές αρκετοί για τις απαιτήσεις της τρέχουσας διργασίας αν ναι η διεργασία δεσμεύευει τους απαραίτητους επεξεργαστές, εκτελείται και ο επεξεργαστής την περιμένει να ολοκληρωθεί για να προχωρήσει στην επόμενη.

```
void fcfs(shared_data*& shptr){
    process proc;
    int status;
    while(true){
        shptr->mtx.lock();
        if(shptr->proc_queue.empty()){
            shptr->mtx.unlock();
            break;
        }
        proc = shptr->proc_queue.front();
        shptr->proc_queue.pop_front();
        if(proc.demands<=shptr->available_processors){
            if(proc.status==PROC_NEW){
                shptr->available_processors-=proc.demands;
                shptr->mtx.unlock();
                proc.t_start = system_clock::now();
                pid_t pid = fork();
                if(pid<0){
                    cerr<<"fork_failed\n";
                    exit(-1);
                }
                else if(pid==0){
                    cout<<"Processor: "<<getppid()<<" executing "<<proc.name<<"\n";
                    execl(proc.name.c_str(), proc.name.c_str(), nullptr);
                }
                else{
                    proc.pid=pid;
                    proc.status=PROC_RUNNING;
                    waitpid(proc.pid, &status, 0);
                    proc.status=PROC_EXITED;
                    proc.t_end=system_clock::now();
                    shptr->mtx.lock();
                    shptr->available_processors+=proc.demands;
                    shptr->mtx.unlock();
                    cout<<"PID "<<pid<< "CMD: "<<proc.name<<"\n";
                    cout<<"\tElapsed time = "<<duration_cast<seconds>(proc.t_end-proc.t_submission).count()<< "secs\n";
                    cout<<"\tExecution time = "<<duration_cast<seconds>(proc.t_end-proc.t_start).count()<<"secs\n";
                    cout<<"\tWorkload time = "<<duration_cast<seconds>(proc.t_end-global_t).count()<<"secs\n";
                }
            }
        }
        else{
            shptr->mtx.unlock();
            if(proc.demands<=num_processors){
                shptr->mtx.lock();
                shptr->proc_queue.push_back(proc);
                shptr->mtx.unlock();
            }
        }
    }
    exit(0);
}
```

Όταν η διεργασία ολοκληρωθεί οι επεξεργαστές που χρησιμοποίησε απελευθερώνονται αυξάνοντας (με ασφαλή τρόπο) την μεταβλητή **available_processors**. Αν δεν υπάρχουν διαθέσιμοι επεξεργαστές για την εκάστοτε διεργασία γίνεται έλεγχος αν οι απαιτήσεις της μπορούν να εκπληρωθούν από το σύστημα. Αν ναι τότε η διεργασία θα επανεισάγεται στο τέλος της ουράς αλλιώς θα απορρίπτεται (καθώς το σύστημα δεν θα μπορούσε ποτέ να την εξυπηρετήσει).

Εκτέλεση του προγράμματος (με το αρχείο reverse.txt του οποίου η δομή έχει παρατεθεί):

```
[alexisl@alex-pc scheduler]$ ./scheduler reverse.txt
Processor: 14854 executing ../work/work7
Processor: 14855 executing ../work/work6
process 14856 begins
process 14858 begins
process 14858 ends
PID 14858CMD: ../work/work6
        Elapsed time = 4secs
        Execution time = 4secs
        Workload time = 4secs
Processor: 14857 executing ../work/work1
process 14861 begins
process 14861 ends
PID 14861CMD: ../work/work1
        Elapsed time = 5secs
        Execution time = 0secs
        Workload time = 5secs
Processor: 14859 executing ../work/work2
process 14862 begins
process 14856 ends
PID 14856CMD: ../work/work7
        Elapsed time = 5secs
        Execution time = 5secs
        Workload time = 5secs
Processor: 14855 executing ../work/work4
Processor: 14857 executing ../work/work5
process 14864 begins
process 14863 begins
process 14862 ends
PID 14862CMD: ../work/work2
        Elapsed time = 7secs
        Execution time = 1secs
        Workload time = 7secs
process 14864 ends
PID 14864CMD: ../work/work4
        Elapsed time = 8secs
        Execution time = 3secs
        Workload time = 8secs
process 14863 ends
PID 14863CMD: ../work/work5
        Elapsed time = 9secs
        Execution time = 4secs
        Workload time = 9secs
Processor: 14859 executing ../work/work3
process 14869 begins
process 14869 ends
PID 14869CMD: ../work/work3
        Elapsed time = 12secs
        Execution time = 2secs
        Workload time = 12secs
WORKLOAD TIME: 12 secs
scheduler exits
[alexisl@alex-pc scheduler]$
```

Όπως είναι φανερό από την έξοδο το σύστημα έχει 4 επεξεργαστές και οι διεργασίες work7, work6 απαιτούν 3 και 1 επεξεργαστές αντίστοιχα, οπότε αρχικά είναι οι μόνες που εκτελούνται ταυτόχρονα και μέχρι κάποια από αυτές να ολοκληρωθεί και να απελευθερώσει τους επεξεργαστές που δέσμευσε δεν εκτελείται καμία άλλη