OPERATING SYSTEMS PROJECT 2

-Κατσαρός Ανδρέας (1084522)
-Ποταμιάνος Άγγελος Νικόλαος (1084537)
-Τζωρτζάκης Γρηγόρης (1084538)

ΑΚΑΔΗΜΑΙΚΟ ΕΤΟΣ: 2022-2023

—

ΕΡΓΑΣΤΗΡΙΟ ΛΕΙΤΟΥΡΓΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

—

ΟΝΟΜΑΤΑ ΚΑΘΗΓΗΤΩΝ:
Κος ΣΠΥΡΙΔΩΝ ΣΙΟΥΤΑΣ
Κος ΧΡΗΣΤΟΣ ΜΑΚΡΗΣ
Κος ΠΑΝΑΓΙΩΤΗΣ ΧΑΤΖΗΔΟΥΚΑΣ
Κος ΑΡΙΣΤΕΙΔΗΣ ΗΛΙΑΣ

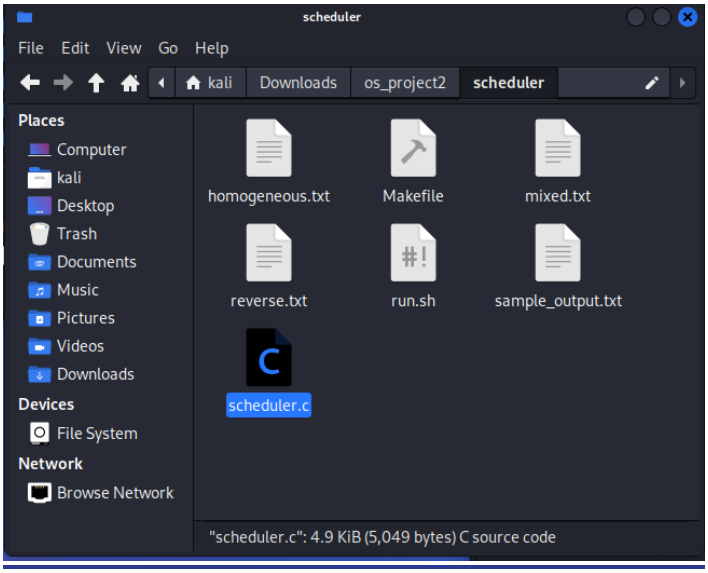
ΕΡΩΤΗΜΑΤΑ:

Σημείωση:

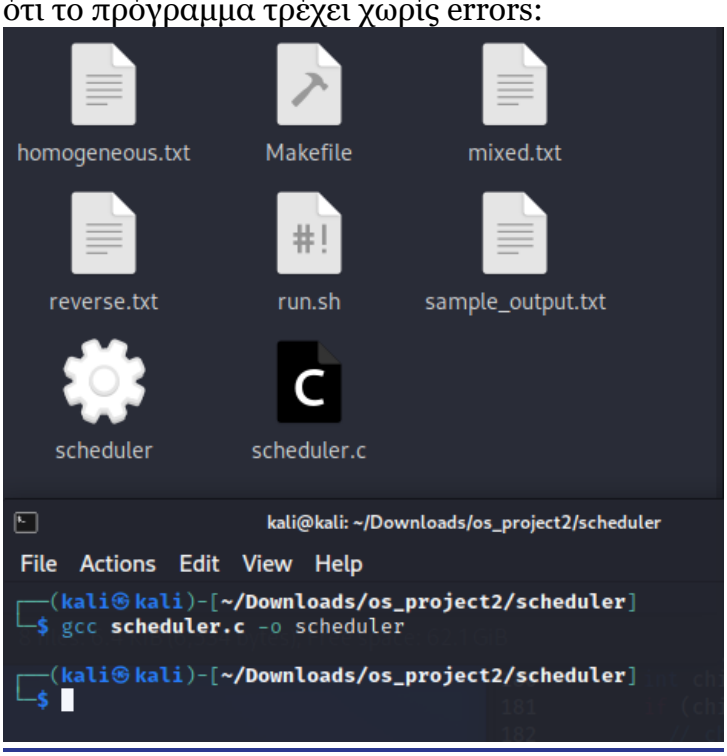
Το παρακάτω πρόγραμμα αποτελεί μια συλλογική προσπάθεια υλοποίησης του πρότζεκτ. Απεικονίζουμε καθαρά τον τρόπο σκέψης μας και δείχνουμε την υλοποίηση του κώδικα. Ο κώδικας αυτός τρέχει χωρίς errors κατά το compile του στο kali , ωστόσο δεν υλοποιεί ορθά τις λειτουργίες που ζητούνται. Προσπαθούμε να απεικονίσουμε την δουλειά μας μέχρι αυτόν τον βαθμό.

ΠΑΡΟΥΣΙΑΣΗ ΣΤΟ VIRTUAL MACHINE:

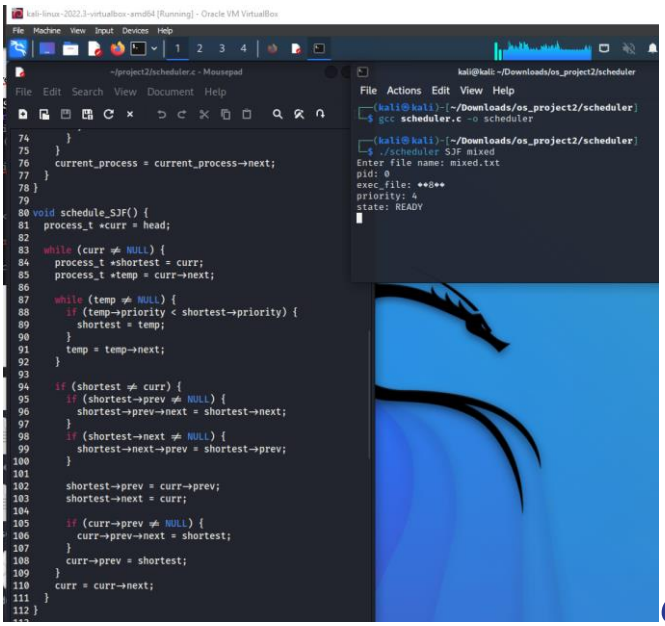
-Ανοίγουμε και τρέχουμε το Virtual BOX
-Έχουμε επιλέξει να εγκαταστήσουμε το Virtual Machine Kali-Linux-2022.3-VirtualBox-amd64
-Ξεκινάμε το Kali και εγκαθιστούμε τα αρχεία του eclass:



-Αφού υλοποιήσαμε το πρόγραμμά μας σε c το τρέχουμε στο terminal με την εντολή και παρατηρούμε ότι το πρόγραμμα τρέχει χωρίς errors:



Τέλος τρέχουμε τον scheduler με ένα από τα αρχεία που έχουμε και παρατηρούμε ένα μέρος του αποτελέσματος:



a

Προκειμένου να υλοποιηθεί το παραπάνω πρότζεκτ απαιτείται η εισαγωγή των παρακάτω βιβλιοθηκών. Συγκεκριμένα, η <sys/wait.h> είναι αναγκαία και υπάρχει στα unix. Παρουσιάζουμε:

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

Δημιουργούμε την δομή Process με σκοπό το διάβασμα των διεργασιών από τα αρχεία που μας δίνονται (φάκελος work). Η υλοποίηση της παραπάνω δομής περιέχει διάφορα είδη μεταβλητών. Αρχικά, δημιουργούμε τα ορίσματα που δίνονται στην εκφώνηση και περιγράφουν την κατάσταση των διεργασιών(όνομα εκτελέσιμου αρχείου, προτεραιότητα, αναγνωριστικό (pid), κατάσταση εκτέλεσης). Ταυτόχρονα, για την προσπάθεια υλοποίησης των χρονοπρογραμματιστικών αλγορίθμων, εισάγουμε τα ορίσματα burst_time(χρόνος εκτέλεσης) wait_time(χρόνος αναμονής) , turn_around_time(χρόνο ολοκλήρωσης). Τέλος ζητείται μία διπλά συνδεδεμένη λίστα (ουρά) στην οποία εισάγονται τα στοιχεία των διεργασιών. Με την χρήση δεικτών, δημιουργούμε τις μεταβλητές struct process *prev, struct process *next που απαιτούνται για την υλοποίηση μιας linked list. Αυτοί, αποθηκεύουν την θέση μνήμης (διεύθυνση) που απαιτείται και όχι το περιεχόμενο του σημείου που δείχνει για την υλοποίηση της λίστας. Έχουμε:

```c
#define MAX_PROCESSES 500


typedef struct process {
  int pid;
  char exec_file[100];
  int priority;
  process_state_t state;
  struct process *prev;
  struct process *next;
  //Aparaithta gia tous algorithmous
  int burst_time;
  int wait_time;
  int turn_around_time;
  //telos edw
} process_t;
```

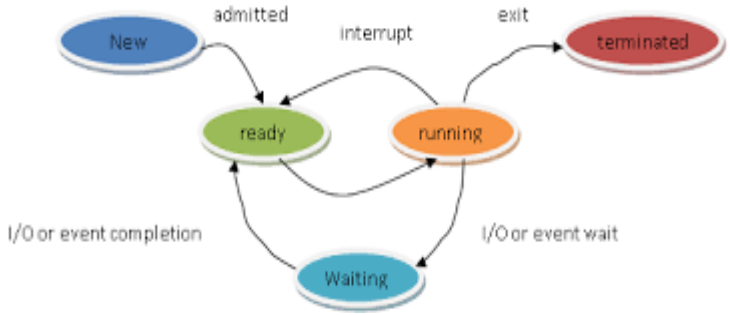
Στη συνέχεια, το πρόγραμμα ξεκινά ζητώντας από τον χρήστη να εισαγάγει το όνομα αρχείου του αρχείου διεργασίας και το ανοίγει με τη εντολή fopen. Εάν το αρχείο δεν υπάρχει, βγάζει μήνυμα σφάλματος. Ταυτόχρονα, η εισαγωγή διεργασιών πραγματοποιείται στο τέλος της λίστας ενώ η εξαγωγή διεργασιών από την αρχή της:

```c
process_t *head = NULL;
process_t *tail = NULL;
```

ΤΡΟΠΟΣ ΣΚΕΨΗΣ ΜΑΣ :

Η συνδεδεμένη λίστα υλοποιείται χρησιμοποιώντας μια δομή δεδομένων λίστας διπλής σύνδεσης, με δείκτες κεφαλής και ουράς. Η συνάρτηση insert_process() προσθέτει μια δομή διεργασίας στο τέλος της συνδεδεμένης λίστας και η συνάρτηση pop_process() αφαιρεί την πρώτη δομή από τη συνδεδεμένη λίστα. Μόλις διαβαστούν και αποθηκευτούν όλες οι διεργασίες, το πρόγραμμα εισάγει έναν βρόχο που συνεχίζεται μέχρι να αδειάσει η λίστα μας. Σε κάθε επανάληψη, το πρόγραμμα αφαιρεί την πρώτη διεργασία από τη λίστα και εκτυπώνει τις πληροφορίες της. Έχουμε την μεταβλητή process_state_t για την υλοποίηση των καταστάσεων εκτέλεσης μιας εφαρμογής (διεργασίας). Εάν η κατάσταση της διεργασίας είναι READY, το πρόγραμμα διαχωρίζει μια διεργασία παιδιού χρησιμοποιώντας τη συνάρτηση exec(). Η γονική διαδικασία περιμένει να ολοκληρωθεί η διεργασία παιδιού με τη λειτουργία waitpid() και ενημερώνει τις πληροφορίες διεργασίας με το αναγνωριστικό και την κατάσταση

θυγατρικής διαδικασίας. Ταυτόχρονα δημιουργούμε τις καταστάσεις RUNNING, STOPPED ή EXITED, στις οποίες απλώς εκτυπώνει την αντίστοιχη κατάσταση. Τέλος, το πρόγραμμα ελευθερώνει τη δομή της διαδικασίας και επαναλαμβάνει τον βρόχο μέχρι να αδειάσει η λίστα.
Με βάση τα process states του Tanenbaum:




```
typedef enum {
  READY,
  RUNNING,
  STOPPED,
  EXITED
} process_state_t;

void insert_process(process_t *process) {
  if (head == NULL) {
    head = tail = process;
  } else {
    tail->next = process;
    process->prev = tail;
    tail = process;
  }
}

process_t *pop_process() {
  process_t *process = head;
  if (head != NULL) {
    head = head->next;
    if (head != NULL) {
      head->prev = NULL;
    }
  }
  return process;
}
```

Παρακάτω στο πρόγραμμα προσπαθήσαμε να εφαρμόσουμε τους τρεις αλγόριθμους προγραμματισμού: First-Come First-Served (FCFS), Shortest Job First (SJF) και Round Robin (RR).

Ξεκινάμε, όπως είπαμε παραπάνω, διαβάζοντας το όνομα ενός αρχείου εισόδου από τον χρήστη. Στη συνέχεια, το αρχείο εισόδου ανοίγει και διαβάζεται για να δημιουργηθεί μια συνδεδεμένη λίστα διεργασιών, όπου κάθε διεργασία έχει ένα PID, όνομα αρχείου εκτέλεσης, προτεραιότητα και κατάσταση. Η συνδεδεμένη λίστα χρησιμοποιείται για τους αλγόριθμους προγραμματισμού.

Γενικά, ο FCFS είναι ένα μη εκτοπιστικός αλγόριθμος ,ο οποίος καλείτε να επιλέξει διεργασίες από το ready queue ώστε να εκτελεστούν(running).Κριτήρια επιλογής του είναι ο χρόνος άφιξης της διεργασίας, δηλαδή θα εκτελεστούν με την σειρά που έφτασαν στο σύστημα. Συγκεκριμένα, εφόσον είναι μη εκτοπιστικός αλγόριθμος, επιλέγει διεργασία να τρέξει μόνο σε δυο περιπτώσεις:
1) Η διεργασία που βρισκόταν σε κατάσταση running βρέθηκε σε κατάσταση stopped καθώς περιμένει δεδομένα για την συνέχιση της εκτέλεσης του.
2)Η διεργασία που βρισκόταν σε κατάσταση ready τελείωσε την εκτέλεση της (terminated).
Δημιουργούμε την συνάρτηση run_fcfs με σκοπό την χρήση του αλγόριθμου. Ο αλγόριθμος προγραμματισμού FCFS ξεκινά επαναλαμβάνοντας τη συνδεδεμένη λίστα διεργασιών. Για κάθε διεργασία που βρίσκεται σε κατάσταση READY, το πρόγραμμα δημιουργεί μια διαδικασία παιδί χρησιμοποιώντας την κλήση συστήματος fork() και εκτελεί τη διαδικασία χρησιμοποιώντας την κλήση συστήματος execl(). Η γονική διαδικασία περιμένει την έξοδο της διαδικασίας παιδί χρησιμοποιώντας την κλήση συστήματος waitpid().Ανάλογα με την κάθε state εκτελεί τις ενέργειες που προαναφέραμε.

```
void run_fcfs() {
  process_t *current_process = head;
  while (current_process != NULL) {
    if (current_process->state == READY) {
      int child_pid = fork();
      if (child_pid == 0) {
        // diadikasia paidi
        execl(current_process->exec_file, current_process->exec_file,
(char *)NULL);
      } else {
        // diadikasia goneas
        current_process->pid = child_pid;
        current_process->state = RUNNING;
        int status;
        waitpid(child_pid, &status, WUNTRACED);
        if (WIFSTOPPED(status)) {
          current_process->state = STOPPED;
        } else {
          current_process->state = EXITED;
        }
      }
```

```
    }
    current_process = current_process->next;
  }
}
```

Ο SJF είναι μη εκτοπιστικός αλγόριθμος ,ο οποίος έχει ως κριτήριο επιλογής την διεργασία με τον μικρότερο χρόνο εκτέλεσης πρώτα. Αναλυτικότερα, αν δυο διεργασίες φτάσουν την ίδια χρονική στιγμή στο σύστημα, συγκρίνει τους χρόνους εκτέλεσης και διαλέγει να εκτελεστεί πρώτα η διεργασία με τον μικρότερο χρόνο. Δημιουργούμε την συνάρτηση schedule_SJF ώστε να χρησιμοποιήσουμε τον αλγόριθμο Ο αλγόριθμος προγραμματισμού SJF αναδιατάσσει τη συνδεδεμένη λίστα διεργασιών έτσι ώστε οι διεργασίες με τη χαμηλότερη τιμή προτεραιότητας να βρίσκονται στην κορυφή της λίστας. Ο αλγόριθμος χρησιμοποιεί έναν απλό αλγόριθμο ταξινόμησης για να αναδιατάξει τη λίστα.

```
void schedule_SJF() {
  process_t *curr = head;

  while (curr != NULL) {
    process_t *shortest = curr;
    process_t *temp = curr->next;

    while (temp != NULL) {
      if (temp->priority < shortest->priority) {
        shortest = temp;
      }
      temp = temp->next;
    }

    if (shortest != curr) {
      if (shortest->prev != NULL) {
        shortest->prev->next = shortest->next;
      }
      if (shortest->next != NULL) {
        shortest->next->prev = shortest->prev;
      }

      shortest->prev = curr->prev;
      shortest->next = curr;

      if (curr->prev != NULL) {
        curr->prev->next = shortest;
      }
      curr->prev = shortest;
    }
    curr = curr->next;
  }
}
```

Ο round robin είναι ένας εκτοπιστικός αλγόριθμος οποίος διαθέτει συγκεκριμένο χρόνο εκτέλεσης (κβάντο) για κάθε διεργασία. Διαλέγει μια διεργασία να εκτελεστεί από το ready queue και όταν αυτή εξαντλήσει τον χρόνο κβάντου, η εκτέλεση της διακόπτεται από την επόμενη διεργασία που βρίσκεται στο συγκεριμένο queue. Εφόσον δεν έχει ολοκληρωθεί , η διεργασία που διακόπτεται επανατοποθετείται στην λίστα με τις έτοιμες διεργασίες και περιμένει ξανά την σειρά της. Εάν ολοκληρωθεί νωρίτερα από τον χρόνο που ορίζει το κβάντο, τότε επιλέγεται η επόμενη διεργασία που περιμένει.
Δεν καταφέραμε να τον υλοποιήσουμε ορθά, ωστόσο προσπαθήσαμε να αναπαραστήσουμε όσο το δυνατόν καλύτερα την λογική του αλγορίθμου. Συγκεκριμένα, ο αλγόριθμος χρονοπρογραμματισμού RR υλοποιείται ως ξεχωριστή συνάρτηση που λαμβάνει ως είσοδο μια σειρά από διεργασίες, τον αριθμό των διεργασιών και την τιμή κβάντου. Ο αλγόριθμος χρησιμοποιεί έναν βρόχο για την προσομοίωση της διαδικασίας προγραμματισμού και παρακολουθεί την τρέχουσα ώρα και τον υπόλοιπο αριθμό διαδικασιών. Ο χρόνος εκτέλεσης κάθε διεργασίας μειώνεται κατά τιμή κβαντου ή κατά τον υπόλοιπο χρόνο ριπής εάν ο χρόνος ριπής είναι μικρότερος από αυτήν. Ο χρόνος αναμονής, ο χρόνος διεκπεραίωσης και ο υπολειπόμενος αριθμός διεργασιών υπολογίζονται και ενημερώνονται. Εδώ, η μεταβλητή wait_time(χρόνος αναμονής) κάθε φορά, παίρνοντας ως όρισμα την στιγμή που υλοποιείται μείον την τιμή που εισάγεται στο πρόγραμμα. Τις απαραίτητες πληροφορίες για την κάθε διεργασία τις παίρνει από το όρισμα processes το οποίο είναι struct τύπου process_t (το οποίο έχει εξηγηθεί). Με αντίστοιχο τρόπο έχουμε τις turn_around_time(χρόνος διεκπεραίωσης) & burst_time(χρόνος εκτέλεσης). Ωστόσο όπως αναφέρθηκε ο παρακάτω υλοποιημένος κώδικας παρουσιάζεται αποκλειστικά για να απεικονίσει τον τρόπο σκέψης μας.

```
void run_RR( process_t processes[], int n, int quantum) {
  int current_time = 0;
  int remaining_processes = n;

  while (remaining_processes) {
    for (int i = 0; i < n; i++) {
      if (processes[i].burst_time > 0) {
        if (processes[i].burst_time > quantum) {
          current_time += quantum;
          processes[i].burst_time -= quantum;
        } else {
          current_time += processes[i].burst_time;
          processes[i].wait_time = current_time - processes[i].burst_time;
          processes[i].turn_around_time = current_time;
          processes[i].burst_time = 0;
          remaining_processes--;
        }
      }
```

```
    }
  }
}
```

Παρακάτω επεξηγούμε την main στην οποία καλούμε τις παραπάνω συναρτήσεις, δημιουργούμε τις κατάλληλες μεταβλητές των αντίστοιχων δομών με σκοπό το καλύτερο επιθυμητό αποτέλεσμα. Πρώτα απ' όλα, αφού διαβάζουμε το αρχείο:

```
 char filename[100];
FILE *fp = fopen(filename, "r");
  if (fp == NULL) {
    printf("Error opening file\n");
    return 1;
  }
```

Εκτυπώνουμε τα στοιχεία που έχουμε ήδη εξηγήσει και αφορούν την δομή τη διεργασίας μας. Ταυτόχρονα , καλούμε τις συναρτήσεις των FCFS και SJF με σκοπό την εξαγωγή τους (output).

```
   process_t *process = pop_process();
    printf("pid: %d\n", process->pid);
    printf("exec_file: %s\n", process->exec_file);
    printf("priority: %d\n", process->priority);
    printf("state: ");

    run_fcfs();


    schedule_SJF();
```

Στη συνέχεια με βάση το αρχείο που εισάγαμε ,ο κώδικας διαβάζει επανειλημμένα 4 τιμές από το αρχείο (pid, exec_file, priority, state_str) έως ότου είτε φτάσει στο τέλος του αρχείου είτε ο αριθμός των διαδικασιών που διαβάστηκαν φτάσει τη μέγιστη τιμή MAX_PROCESSES. Για κάθε σύνολο τιμών που διαβάζεται, δημιουργείται μια νέα δομή process_t χρησιμοποιώντας malloc και οι τιμές αποθηκεύονται στη δομή. Η συμβολοσειρά κατάστασης συγκρίνεται με 4 πιθανές τιμές (READY, RUNNING, STOPPED, EXITED) και το πεδίο κατάστασης της δομής ορίζεται ανάλογα. Τέλος, η διαδικασία που δημιουργήθηκε πρόσφατα προστίθεται σε μια λίστα διεργασιών χρησιμοποιώντας τη συνάρτηση insert_process και ο βρόχος συνεχίζεται. Στη συνέχεια, το αρχείο κλείνει.

```
   int pid, priority;
  char exec_file[100];
  char state_str[20];
  int count = 0;
  while (fscanf(fp, "%d %s %d %s", &pid, exec_file, &priority, state_str)
!= EOF && count < MAX_PROCESSES) {
    process_t *process = (process_t *)malloc(sizeof(process_t));
    process->pid = pid;
    strcpy(process->exec_file, exec_file);
    process->priority = priority;
```

```
    if (strcmp(state_str, "READY") == 0) {
      process->state = READY;
    } else if (strcmp(state_str, "RUNNING") == 0) {
      process->state = RUNNING;
    } else if (strcmp(state_str, "STOPPED") == 0) {
      process->state = STOPPED;
    } else if (strcmp(state_str, "EXITED") == 0) {
      process->state = EXITED;
    }
    insert_process(process);
    count++;
  }
  fclose(fp);
```

Τρόπος σκέψης:
Αυτός ο κώδικας εκτελεί τις ακόλουθες ενέργειες:

Ανακτά μια διεργασία από το μπροστινό μέρος της ουράς χρησιμοποιώντας τη συνάρτηση pop_process() που εξηγήσαμε.

Εκτυπώνει το pid, το exec_file, την προτεραιότητα και την κατάσταση της διαδικασίας.

Εκτελεί μια δήλωση switch για την κατάσταση της.

Εάν η διεργασία είναι READY, δημιουργεί μια νέα διεργασία παιδί χρησιμοποιώντας την κλήση συστήματος fork() και εκτελεί το exec_file της διαδικασίας χρησιμοποιώντας τη συνάρτηση execl().

Εάν είναι η γονική διεργασία, περιμένει να ολοκληρωθεί η διεργασία παιδιού χρησιμοποιώντας τη συνάρτηση waitpid() και ενημερώνει την κατάσταση της διαδικασίας είτε σε STOPPED είτε σε EXITED.

Στη συνέχεια, εισάγει τη διεργασία πίσω στην ουρά χρησιμοποιώντας τη συνάρτηση insert_process().

Εάν η διεργασία εκτελείται, απλώς εκτυπώνει την κατάσταση.

Εάν η διεργασία είναι STOPPED, εκτυπώνει την κατάσταση.

Εάν η διεργασία είναι EXITED, εκτυπώνει την κατάσταση.

Έπειτα, ελευθερώνει τη μνήμη που έχει εκχωρηθεί στη διαδικασία.

```
  process_t *process = pop_process();
   printf("pid: %d\n", process->pid);
   printf("exec_file: %s\n", process->exec_file);
   printf("priority: %d\n", process->priority);
   printf("state: ");
   switch (process->state) {
     case READY:
       printf("READY\n");
       int child_pid = fork();
       if (child_pid == 0) {
         // child process
         execl(process->exec_file, process->exec_file, (char *)NULL);
       } else {
         // parent process
         process->pid = child_pid;
         process->state = RUNNING;
         int status;
         waitpid(child_pid, &status, WUNTRACED);
```

```
          if (WIFSTOPPED(status)) {
            process->state = STOPPED;
          } else {
            process->state = EXITED;
          }
          insert_process(process);
        }
        break;
      case RUNNING:
        printf("RUNNING\n");
        break;
      case STOPPED:
        printf("STOPPED\n");
        break;
      case EXITED:
        printf("EXITED\n");
        break;
    }
```

Επιπλέον, ελευθερώνει και τη μνήμη που έχει δεσμευθεί για την υλοποίηση της process με την συνάρτηση free (σε υπάρχουσα βιβλιοθήκη) που έχει εκχωρηθεί στη διαδικασία και κλείνουμε την main.

```
free(process);

return 0;

}
```

Στο τέλος της αναφοράς επισημαίνεται ο δρομολογητής επιλέγει την επόμενη διεργασία που θα εκτελείται κάθε φορά και εάν υπάρχουν διεργασίες ,εκτελεί τις κατάλληλες ενέργειες διαφορετικά τερματίζει. Θεωρούμε ότι οι διαδικασίες που θα δρομολογηθούν περιλαμβάνουν αποκλειστικά υπολογισμούς και καθόλου εντολές εισόδου - εξόδου.
Συνεπώς, στην περίπτωση των πολιτικών στατικής δρομολόγησης (FCFS, SJF), ο δρομολογητής θα ενεργοποιείται μόνο όταν μια διαδικασία τερματίζεται. Στην κατηγορία των πολιτικών δρομολόγησης, η διαχείριση της εκτέλεσης διαδικασιών, θα γίνει με τη χρήση των σημάτων SIGSTOP και SIGCONT. Τέλος, όταν μια διεργασία τερματίζεται, θα πρέπει να ειδοποιηθεί ο δρομολογητής, έχοντας ρυθμίσει χειριστή για το σήμα SIGCHLD και εκτέλεση των απαραίτητων ενεργειών για τη σωστή λειτουργία του.
Το παραπάνω ζητούμενο δεν το υλοποιήσαμε στον κώδικα λόγω δυσκολίας. Ωστόσο μέσω διάφορων forum και αναζήτησης που κάναμε βρήκαμε και παραθέτουμε την παρακάτω ενδεικτική λύση για την υλοποίηση του ζητούμενου:

Ενδεικτικές πηγές: StackOverflow, w3school.

```
//ENDEIKTIKH YLOPOIHSH SIGCHILD
// SIGCHLD signal handler to update the router and perform necessary
actions when an application terminates
void sigchld_handler(int sig) {
  int status;
  process_t pid = waitpid(-1, &status, WNOHANG);
  if (pid == 0) {
    return;
  }
  for (int i = 0; i < process_count; i++) {
if (process_t[i].pid == pid) {
time_t end_time = time(NULL);
int execution_time = end_time - process_t[i].turn_around_time;
printf("Process %d completed in %d seconds\n", process_t[i].pid,
execution_time);
}
}
running = 0;
}
```

ΕΥΧΑΡΙΣΤΟΎΜΕ ΓΙΑ ΤΟΝ ΧΡΟΝΟ ΣΑΣ