Λειτουργικά Συστήματα

2022 - 2023

2η Εργαστηριακή Άσκηση

Η εργασία υλοποιήθηκε από τούς:

- **Γρηγόρης Δελημπαλταδάκης**, ΑΜ**: 1084647**, email: **up1084647@upnet.gr**
- **Δαμιανός Διασάκος**, ΑΜ**: 1084632**, email: **up1084632@upnet.gr**
- **Αλκιβιάδης Δασκαλάκης**, ΑΜ**:** email: **1084673**, **up1084673@upnet.gr**
- **Ιάσων Ράικος**, ΑΜ**: 1084552**, email: **up1084552@upnet.gr**

ΠΕΡΙΕΧΟΜΕΝΑ

1. Απαραίτητες ενέργειες πριν τρέξουμε το πρόγραμμα. 2

2. Επεξήγηση του ‘scheduler.c’. 2

1.1 Απαραίτητα στοιχεία 2

1.2 Αλγόριθμοι 8

1.3 Main 23

1.4 Αποτελέσματα εκτέλεσης του run.sh 28

1. Απαραίτητες ενέργειες πριν τρέξουμε το πρόγραμμα.

1. Πρέπει οι 2 φάκελοι '**scheduler'** και '**work'** να βρίσκονται μέσα στον ίδιο φάκελο για να μπορεί να εκτελεστεί η εντολή **cd ../work/workN, N={1,2,3,4,5,6,7}.**

2. Είναι απαραίτητο στον φάκελο **work** να εκτελέσουμε την εντολή '**make'** έτσι ώστε να δημιουργηθούν τα εκτελέσιμα αρχεία '**work1'**, '**work2'**,' **work3'**, '**work4'**, '**work5'**, '**work6'** και '**work7'**.

3. Compilation του αρχείου 'scheduler.c' με την εντολή '**gcc scheduler.c -o scheduler**'.

2. Επεξήγηση του 'scheduler.c'.

1.1 Απαραίτητα στοιχεία

- Struct Work που θα περιέχει τα στοιχεία κάθε διεργασίας όπως τον αριθμό N που αντιπροσωπεύει την διεργασία, το pid της, το command που είναι της μορφής '**../work/workN**' ,τον χρόνο που ολοκληρώθηκε, τα pointers που θα χρειάζονται για την ουρά αναμονής καθώς και 2 μεταβλητές status και exited για την περίπτωση των δυναμικών πολιτικών δρομολόγησης. Επιπλέον έχουμε και ένα index για να κρατάμε την σειρά που ολοκληρώνονται οι διεργασίες.

```
#define MAX_LEN_COMMAND 15

struct Work

{

    int number; //o arithmos tou executable workN

    int priority;

    pid_t pid;

    double time; //elapsed time

    double workload_rr; // workload time

    double workload_prio; // workload time

    struct timeval start_time, end_time; ////gia ne metrisw ton xrono ektelesis kathe
diergasias

    char command[MAX_LEN_COMMAND];

    struct Work *next; //pointer gia to queue

    struct Work *prev;

    int status;

    bool exited;

    int index; //gia tin ektypwsi me tin seira pou oloklirwnontai oi diergasies
```

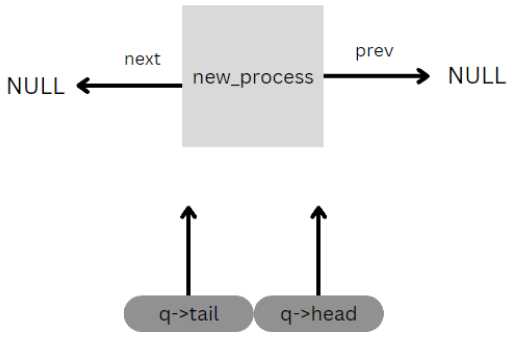
- Ουρά αναμονής με διπλή συνδεδεμένη λίστα όπου περιέχει pointer για την κεφαλή και την ουρά. Επίσης, έχει δημιουργηθεί συνάρτηση που αρχικοποιεί την ουρά αναμονής.

```
struct WorkQueue
{
    struct Work *head;
    struct Work *tail;
};

void init_queue(struct WorkQueue *q)
{
    q->head = NULL;
    q->tail = NULL;
}
```

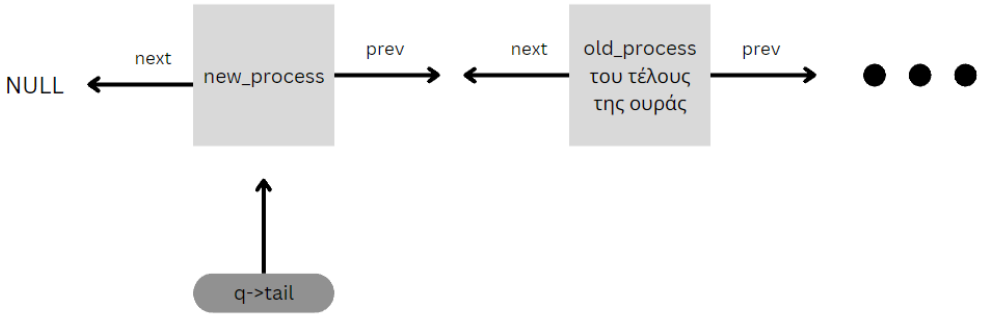
- Συνάρτηση catch_sigchld η οποία με βάση την παράμετρο status της κάθε διεργασίας ελέγχει αν έχει τερματίσει η διεργασία και θέτει τις τιμές exited και time κατάλληλα.

```
void catch_sigchld(int sig)
{
    waitpid(current_process->pid, &current_process->status, WNOHANG);
    if(WIFEXITED(current_process->status))
    {
        current_process->exited = true;
        gettimeofday(&current_process->end_time, NULL);
        current_process->time = (current_process->end_time.tv_sec - current_process->start_time.tv_sec) +
        (current_process->end_time.tv_usec - current_process->start_time.tv_usec) / 1000000.0;
        current_process->workload_rr = (current_process->end_time.tv_sec-start_rr.tv_sec)+(current_process->end_time.tv_usec-start_rr.tv_usec)/ 1000000.0;
        current_process->workload_prio = (current_process->end_time.tv_sec-start_prio.tv_sec)+(current_process->end_time.tv_usec-start_prio.tv_usec)/ 1000000.0;  }}
```

- Υλοποίηση συνάρτησης για εισαγωγή δεδομένων στο τέλος της ουράς αναμονής. Πριν εισαχθεί κάποιο process στην ουρά οι δείκτες head και tail είναι NULL. Σε αυτή την περίπτωση με την εισαγωγή της πρώτης εργασίας θέλουμε και οι δυο δείκτες να δείχνουν στην καινούρια εργασία.

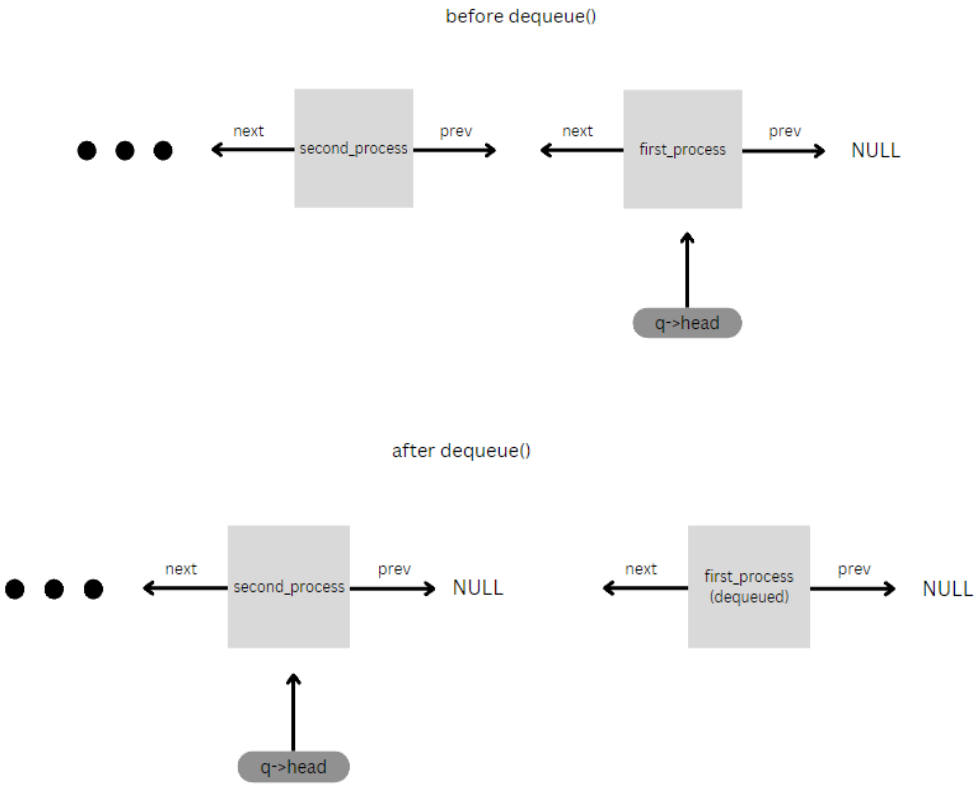


Αλλιώς εισάγουμε την νέα εργασία πίσω από την εργασία στην οποία έδειχνε προηγουμένως ο δείκτης tail της ουράς.

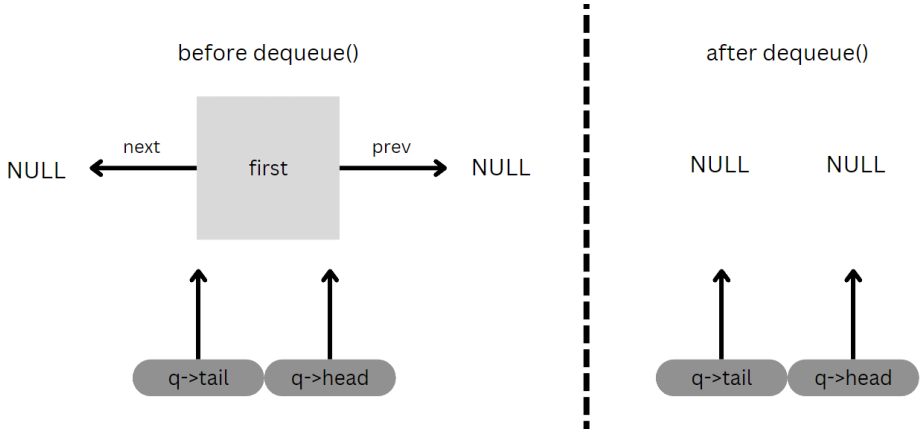


```
void enqueue(struct WorkQueue *q, struct Work *new_process)
{


   new_process->prev = NULL;
   new_process->next = NULL;


   if (q->tail == NULL)
   {
      q->head = new_process;
      q->tail = new_process;
   }
   else
   {
      q->tail->next = new_process;
      new_process->prev = q->tail;
      q->tail = new_process;
   }
}
```

- Υλοποίηση εξαγωγής δεδομένων από την ουράς αναμονής. Αν η ουρά αναμονής είναι άδεια τότε θα επιστραφεί null. Διαφορετικά η συνάρτηση επιστρέφει έναν pointer σε struct τύπου Work το οποίο θέλουμε να είναι το process της κεφαλής της ουράς. Έτσι εκτελούμε την εντολή struct Work *first_process = q->head και κάνουμε return τον δείκτη, στο τέλος της συνάρτησης. Για να επιστρέψουμε την ουρά τώρα σε σωστή κατάσταση λειτουργίας θέτουμε τον δείκτη head να δείχνει στην προηγούμενη εργασία αυτής που μόλις εξηγάγαμε. Πρέπει όμως να διακρίνουμε περιπτώσεις.
Αν ο δείκτης head τώρα δεν είναι NULL σημαίνει ότι δεν εξηγάγαμε το τελευταίο στοιχείο από την ουρά και θέτω απλά το prev pointer της καινούριας κεφαλής/process ίσο με NULL.



Αλλιώς αν ο δείκτης head γίνει NULL σημαίνει πως εξηγάγαμε την τελευταία εργασία της ουράς οπότε πρέπει και ο tail pointer να γίνει NULL.



```
struct Work *dequeue(struct WorkQueue *q)
{
    if (q->head == NULL)
    {
        return NULL;
    }
    struct Work *first_process = q->head;
    q->head = first_process->next;


    // kanoniki katastasi
    // an meta tin afairesi tou prwtou process apo tin lista/queue o deiktis q->head deixnei kapou tote thetw ton deikti
    // prev tou kainouriou head process ws null
    if (q->head != NULL)
    {
        q->head->prev = NULL;
    }


    // an meta tin afairesi tou prwtou process apo tin lista o deiktis q->head ginei NULL simainei oti ekana dequeue to
    // telautaio process opote to q->tail edeixne ekei pou edeixne arxika kai o q->head opote prepei kai autos na ginei NULL
    if (q->head == NULL)
    {
        q->tail = NULL;
    }
    return first_process;
}
```

1.2 Αλγόριθμοι

- **FCFS(FIRST COME FIRST SERVED)** ή **BATCH** είναι ο αλγόριθμος ο οποίος εκτελεί κάθε διεργασία με την σειρά που έρχονται. Άρα, οι διεργασίες θα εισέλθουν στην ουρά με την σειρά που είναι γραμμένες και με αυτήν θα εκτελεστούν.

```
void FCFS(int process_count, struct WorkQueue q, struct Work works[])
{
   for (int i = 0; i < process_count; i++)
   {
      struct Work *first_work = dequeue(&q);

      int pid = fork();
      if (pid == 0)
      {
         // This is the child process
         char *args[] = {first_work->command, NULL};
         execvp(first_work->command, args);
         exit(0);
      }
      else if (pid > 0)
      {
         // This is the parent process
         struct timeval start, end;
         gettimeofday(&start, NULL);
         waitpid(pid, NULL, 0);
         gettimeofday(&end, NULL);
         works[i].pid = pid;
         works[i].time = (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) / 1000000.0;
      }
   }
}
```

Σαν παραμέτρους έχει μία μεταβλητή **process_count**, η οποία δηλώνει τον αριθμό των διεργασιών, ένα **struct q** τύπου **WorkQueue** για να γίνει χρήση της ουράς και ένα **struct works** τύπου **Work** που δηλώνει το **struct** των διεργασιών. Γίνεται χρήση ενός βρόγχου για να χρησιμοποιούμε κάθε φορά το **command** στην **execvp** κάθε διεργασίας που είναι στην αρχή της ουράς. Για να έχουμε πρόσβαση στην $1^{η}$ διεργασία στην ουρά καλούμε την συνάρτηση **dequeue()** όπου την αναθέτουμε σε ένα ***struct** τύπου **Work** ονόματι **first_work**. Επίσης χρησιμοποιούνται οι συναρτήσεις **gettimeofday()** για να καταφέρουμε να υπολογίσουμε τον χρόνο που ξεκίνησε και τελείωσε η διεργασία, βρίσκοντας έτσι και τον συνολικό της χρόνο.

Επιπλέον, χρησιμοποιούμε την **fork()** για να δημιουργήσουμε ένα **child** της διεργασίας έτσι ώστε να μπορέσουμε να εκτελέσουμε παράλληλα το **command** αλλά και να μπορέσουμε να έχουμε πρόσβαση στο **pid** της διεργασίας **father**. Γίνεται και χρήση της **waitpid()** η οπόια «περιμένει» να ολοκληρωθεί η διεργασία **child** έτσι ώστε να καταφέρουμε να υπολογίσουμε τον χρόνο που χρειάστηκε για την διεργασία.

Τέλος, αναθέτουμε στο **struct works** τύπου **Work** τις τιμές του **pid** και του χρόνου έτσι ώστε να μπορούμε να τα εκτυπώσουμε αργότερα στην **main**.

- Όσον αφορά τον αλγόριθμο **SJF**(Shortest Job First) κάθε εφαρμογή χαρακτηρίζεται από έναν στατικό χρόνο εκτέλεσης (**αριθμό**), ο οποίος καθορίζει τη σειρά εκτέλεσής της. Οι εφαρμογές με το μικρότερο αριθμό έχουν τη μεγαλύτερη προτεραιότητα. Στα αρχεία κειμένου όπως για πάραδειγμα το reverse.txt αυτό ο **αριθμός** είναι ο αριθμός Ν της κάθε work που συμβολίζει το delay.
  reverse.txt:
  ../work/work**7** 7
  ../work/work**6** 6
  ../work/work**5** 5
  ../work/work**4** 4
  ../work/work**3** 3
  ../work/work**2** 2
  ../work/work**1** 1

  Οπότε, σαν αλγόριθμος ο **SJF** «πάνω» από την main έχει υλοποιηθεί ακριβώς σαν τον **FCFS** με μόνη διαφορά ότι στο **struct processes** πραγματοποιούμε ταξινόμηση των διεργασιών βάση τον αριθμό τους Ν προτεραιότητας τους (**number**). Αυτό γίνεται διότι ο **SJF** εκτελεί πρώτα τις εντολές που διαρκούν το λιγότερο χρονικό διάστημα. Όμως αφού δεν μπορούμε να γνωρίζουμε από πριν τον ακριβή χρόνο που θα τρέξουν αλλά γνωρίζουμε το κάθε **delay** που θα έχει η κάθε μία χάρις το **work.c** και το **makefile** σαν αριθμό για χρονική διάρκεια αναθέσαμε το Ν του κάθε workN. Αφού ταξινομηθούν σωστά, κάνουμε populate την ουρά με την σωστή σειρά την οποία θα αξιοποιήσει ο **SJF**. Για την ταξινόμηση χρησιμοποιήθηκε Insertion Sort ο οποίος έχει υλοποιηθεί παρακάτω.

```
struct Work temp;
    for (int i = 1; i < process_count; i++)
    {
        temp.priority = works[i].priority;
        temp.pid = works[i].pid;
        temp.time = works[i].time;
        temp.number = works[i].number;
        strcpy(temp.command, works[i].command);
        temp.next = works[i].next;
        temp.prev = works[i].prev;
        int j = i - 1;
        while (temp.number < works[j].number && j >= 0)
        {
            works[j + 1].priority = works[j].priority;
            works[j + 1].pid = works[j].pid;
            works[j + 1].time = works[j].time;
            works[j + 1].number = works[j].number;
            strcpy(works[j + 1].command, works[j].command);
            works[j + 1].next = works[j].next;
            works[j + 1].prev = works[j].prev;
            --j;
        }
        works[j + 1].priority = temp.priority;
        works[j + 1].pid = temp.pid;
        works[j + 1].time = temp.time;
        works[j + 1].number = temp.number;
        strcpy(works[j + 1].command, temp.command);
        works[j + 1].next = temp.next;
        works[j + 1].prev = temp.prev;
    }

for (int i = 0; i < process_count; i++)
    {
        enqueue(&q, &works[i]);
    }
```

- **RR(Round Robin)**

  Σύμφωνα με αυτή την πολιτική χρονοπρογραμματισμού κάθε διεργασία της ουράς εκτέλεσης εκτελείται για ένα προκαθορισμένο χρονικό διάστημα (κβάντο) στο τέλος του οποίου, εάν δεν έχει τερματίσει, προστίθεται στο πίσω μέρος της ουρά.

```
void RR(int quantum, struct WorkQueue *q, int success[],int index)
{
   gettimeofday(&start_rr, NULL);
   struct timespec sleep_time;
   if(quantum>=1000)
   {
      sleep_time.tv_sec = quantum/1000;
      sleep_time.tv_nsec = 0;
   }
   else
   {
      sleep_time.tv_sec = 0;
      sleep_time.tv_nsec = quantum * 1000000; //quantum ms = quantum * 10^6 ns
   }

   //gia na pigainei sto catch_sigchld otan kanei mono exit() ena paidi
   struct sigaction sa;
   sa.sa_handler = catch_sigchld;
   sigemptyset(&sa.sa_mask);
   sa.sa_flags = SA_NOCLDSTOP;

   sigaction(SIGCHLD, &sa, NULL); //kanw register tin catch_sigchld function ws handler for the SIGCHLD signal
```

```
while (q->head != NULL)
  {
    current_process = dequeue(q);
    if(WIFSTOPPED(current_process->status))
    {
      kill(current_process->pid, SIGCONT);
    }
    else if(current_process->pid == -1)
    {
      current_process->pid = fork();
      gettimeofday(&current_process->start_time, NULL);
    }

    if (current_process->pid == 0)
    {
      // child process
      char *args[] = {current_process->command, NULL};
      execvp(current_process->command, args);
      exit(0);
    }
    //an kanw dequeu process pou exei teleiwsei den prepei na ksana kanw sleep()
    else if(current_process->exited){
      success[index] = current_process->index;
      index++;
      continue;
    }
```

```
// parent process
        nanosleep(&sleep_time, NULL); // sleep gia xrono iso me to quantum


        if(current_process->exited)
        {
           success[index] = current_process->index;
           index++;
           continue;
        }
        else
        {
           kill(current_process->pid, SIGSTOP);
           waitpid(current_process->pid, &current_process->status, WSTOPPED);
           enqueue(q,current_process);
        }


     }
   }
   gettimeofday(&end_rr, NULL);
}
```

Σε ένα loop που επαναλαμβάνεται όσο υπάρχουν εργασίες στην ουρά εκτέλεσης, κάνουμε **dequeue()** το πρώτο **process**. Πρέπει να σημειωθεί ότι ο **pointer current_process** είναι **global μεταβλητή**, γιατί πρέπει να έχουμε πρόσβαση σε αυτόν και στον signal handler **catch_sigchld()**.
Αν η τρέχουσα διαδικασία ήταν σταματημένη την συνεχίζουμε στέλνοντας σήμα **SIGCONT** με την εντολή **kill(current_process->pid, SIGCONT)**. Για να ελέγξουμε αν μια διεργασία είναι σταματημένη πρέπει το πεδίο **status** του current_process να έχει ενημερωθεί με την εντολή **waitpid(current_process->pid, &current_process->status, WSTOPPED),** ώστε η συνάρτηση **WIFSTOPPED(current_process->status)** μέσα στο if να επιστρέψει true.

Child process

Αν το **current_process** δεν έχει εκτελεστεί καθόλου, δηλαδή έχει το πεδίο pid = -1 (έτσι έχει αρχικοποιηθεί στην main), καλούμε την **fork()** και δημιουργείται ένα child process στο οποίο καλούμε την **execvp(current_process->command)** για να εκτελεστεί το **workn** που αντιστοιχεί στο τρέχον process. Παράλληλα ξεκινάμε το timer του χρόνου εκτέλεσης της συγκεκριμένης διεργασίας με την εντολή **gettimeofday(&current_process->start_time, NULL)**.

Parent process

Στο block κώδικα του parent process καλούμε την nanosleep() για χρονικό διάστημα ίσο με **quantum** ms, για να δώσουμε τον χρόνο στην διεργασία παιδί να εκτελεστεί. Υπάρχει όμως περίπτωση η διεργασία παιδί να τερματίσει πριν το τελειώσει το nanosleep. Σε αυτή την περίπτωση το παιδί στέλνει σήμα SIGCHLD στον πατέρα και καλείται ο **signal handler**, **catch_sigchld(int sig)**. Θέλουμε δηλαδή να καλείται ο **signal handler** που δημιουργήσαμε μόνο όταν η διεργασία – παιδί έχει τερματίσει και όχι κάθε φορά που καλούμε την συνάρτηση **kill()**, γιατί και η κλήση αυτής της συνάρτησης κάνει το παιδί να στείλει σήμα **SIGCHLD**.
Ορίζουμε λοιπόν τον signal handler ως εξής:

```
struct sigaction sa;

sa.sa_handler = catch_sigchld;

sigemptyset(&sa.sa_mask);

sa.sa_flags = SA_NOCLDSTOP;


sigaction(SIGCHLD, &sa, NULL);
```

Στην **catch_sigchld(int sig)** ελέγχουμε αν όντως έχει τερματίσει το παιδί και θέτουμε το boolean πεδίο **exited** της τρέχουσας διεργασίας ίσο με true. Για να ελέγξουμε αν έχει τερματίσει ένα process πρέπει να ενημερώσουμε το πεδίο status **waitpid(current_process->pid, &current_process->status, WNOHANG)** και να χρησιμοποιήσουμε την **WIFEXITED(current_process->status)** μέσα σε ένα if block. Επιπλέον σταματάμε το timer της διεργασίας και υπολογίζουμε τους χρόνους εκτέλεσης.

```
void catch_sigchld(int sig)
{
   waitpid(current_process->pid, &current_process->status, WNOHANG);
   if(WIFEXITED(current_process->status))
   {
      current_process->exited = true;
      gettimeofday(&current_process->end_time, NULL);
      current_process->time = (current_process->end_time.tv_sec - current_process->start_time.tv_sec) +
      (current_process->end_time.tv_usec - current_process->start_time.tv_usec) / 1000000.0;
      current_process->workload_rr = (current_process->end_time.tv_sec-start_rr.tv_sec)+(current_process->end_time.tv_usec-start_rr.tv_usec)/ 1000000.0;
      current_process->workload_prio = (current_process->end_time.tv_sec-start_prio.tv_sec)+(current_process->end_time.tv_usec-start_prio.tv_usec)/ 1000000.0;
    }
}
```

Στην συνέχεια ελέγχουμε το πεδίο **exited** και αν είναι true, δηλαδή αν έχει τελειώσει η εκτέλεση της τρέχουσας διεργασίας, κάνουμε **continue** για να συνεχίσει η εκτέλεση της επόμενης διεργασίας στην ουρά.
Εάν η τρέχουσα διεργασία δεν έχει τερματίσει, σταματάμε την εκτέλεση της, στέλνοντας σήμα **SIGSTOP** με την εντολή **kill(current_process->pid, SIGSTOP)** και ενημερώνουμε το status, **waitpid(current_process->pid, &current_process->status, WSTOPPED)**. Τέλος κάνουμε **enqueue()** τη τρέχουσα διεργασία για να εκτελεστεί ξανά όταν είναι και πάλι η σειρά της.

Μέτρηση χρόνου εκτέλεσης

Για να μετρήσουμε τον **χρόνο εκτέλεσης (elapsed time)** κάθε διεργασίας χρησιμοποιούμε τα πεδία **struct timeval start_time, end_time** του Work struct σε συνδυασμό με την συναρτήση **gettimeofday()** και για εκκίνηση και τερματισμού του timer. Για την μέτρηση του **workload time**, δηλαδή τον χρόνο από την αρχή της εκτέλεσης του αλγορίθμου μέχρι τον τερματισμό της κάθε διεργασίας χρησιμοποιούμε την **global** μεταβλητή **struct timeval start_rr** την οποία ξεκινάμε στην αρχή της συνάρτησης RR(). Αντίστοιχα και για τον **PRIO()** με την **struct timeval start_prio.** Το παραπάνω struct και συνάρτηση ορίζονται στην βιβλιοθήκη **sys/time.h**. Κάθε φορά που τερματίζουμε το timer, αποθηκεύουμε στα πεδία **time, workload_rr, workload_prio** της τρέχουσας διεργασίας το χρόνο που πέρασε ως εξής:

- current_process->time = (current_process->end_time.tv_sec - current_process->start_time.tv_sec) + (current_process->end_time.tv_usec - current_process->start_time.tv_usec) / 1000000.0.
- current_process->workload_rr = (current_process->end_time.tv_sec-start_rr.tv_sec)+(current_process->end_time.tv_usec-start_rr.tv_usec)/ 1000000.0.
- current_process->workload_prio = (current_process->end_time.tv_sec-start_prio.tv_sec)+(current_process->end_time.tv_usec-start_prio.tv_usec)/ 1000000.0.

- PRIO

```
void PRIO(int process_count, struct WorkQueue q, struct Work works[],int quantum,int success[])
{
   struct Work temp[process_count];
   for (int i = 1; i < process_count; i++)
   {
      temp[0].priority=works[i].priority;
      temp[0].pid=works[i].pid;
      temp[0].time=works[i].time;
      temp[0].number=works[i].number;
      strcpy(temp[0].command,works[i].command);
      temp[0].next=works[i].next;
      temp[0].prev=works[i].prev;
      int j=i-1;

      while (temp[0].priority < works[j].priority && j >= 0)
      {
         works[j + 1].priority=works[j].priority;
         works[j + 1].pid=works[j].pid;
         works[j + 1].time=works[j].time;
         works[j + 1].number=works[j].number;
         strcpy(works[j + 1].command,works[j].command);
         works[j + 1].next=works[j].next;
         works[j + 1].prev=works[j].prev;
         --j;
      }
      works[j + 1].priority=temp[0].priority;
      works[j + 1].pid=temp[0].pid;
      works[j + 1].time=temp[0].time;
      works[j + 1].number=temp[0].number;
      strcpy(works[j + 1].command,temp[0].command);
      works[j + 1].next=temp[0].next;
      works[j + 1].prev=temp[0].prev;
   }
```

```
for (int i=0; i<process_count; i++)
    {
        enqueue(&q,&works[i]);
    }
    bool found_same_priority=false;
    int start_of_same_priority=0;
    struct Work *prev_work=NULL;
    struct Work *first_work=NULL;
    struct WorkQueue temp_q;
    init_queue(&temp_q);
    gettimeofday(&start_prio, NULL);
    first_work=dequeue(&q);
    for (int i=0; i<process_count; i++)
    {
        int pid=-1;
        int temp_process_position=0;
        prev_work=first_work;
        first_work=dequeue(&q);

        if(first_work==NULL && found_same_priority==false)
        {
            goto last_work_not_in_same_priority_queue;
        }
        else if(first_work==NULL && found_same_priority==true)
        {
            goto last_work_in_same_priority_queue;
        }
        if(first_work->priority==prev_work->priority)
        {
            found_same_priority=true;

            continue;
        }
```

```
else
    {
      if(found_same_priority==true)
      {
        last_work_in_same_priority_queue:
        temp_process_position=start_of_same_priority;
        for(int x=0; x<i-start_of_same_priority+1; x++)
        {
          temp[x]=works[temp_process_position];
          temp_process_position++;
        }
        for (int k=0; k<i-start_of_same_priority+1; k++)
        {
          enqueue(&temp_q,&temp[k]);
        }
        RR(quantum,&temp_q,success,start_of_same_priority);
        int process_position=start_of_same_priority;
        for(int d=0; d<i-start_of_same_priority+1; d++)
        {
          works[process_position]=temp[d];
          process_position++;
        }
        found_same_priority=false;
        start_of_same_priority=i+1;
        continue;
      }
      start_of_same_priority=i+1;
    }
```

```
last_work_not_in_same_priority_queue:


        pid=fork();
        if (pid==0)
        {
            // This is the child process
            char *args[]={prev_work->command,NULL};
            execvp(prev_work->command,args);
            exit(0);
        }
        else if (pid>0)
        {
            // This is the parent process
            struct timeval start,end;
            gettimeofday(&start,NULL);
            waitpid(pid, NULL,0);
            gettimeofday(&end,NULL);
            works[i].pid=pid;
            works[i].time=(end.tv_sec-start.tv_sec)+(end.tv_usec-start.tv_usec)/1000000.0;
            works[i].workload_prio=(end.tv_sec-start_prio.tv_sec)+(end.tv_usec-start_prio.tv_usec)/1000000.0;
            success[i]=i;
        }
    }
    gettimeofday(&end_prio, NULL);
}
```

Σαν παραμέτρους έχει μία μεταβλητή **process_count**,η οποία δηλώνει τον αριθμό των διεργασιών,ένα **struct q** τύπου **WorkQueue** για να γίνει χρήση της ουράς,ένα πίνακα **works** τύπου **struct Work** που δηλώνει το **struct** των διεργασιών,μία μεταβλητή **quantum** που έχει την ίδια χρήση όπως στον **RR**(**Round Robin**) και ένα πίνακα **success** ο οποίος κρατά με την σειρά τις θέσεις των διεργασιών πάνω στον πίνακα **works** που τελείωσαν πρώτες.

Στη συνέχεια, κάνουμε τον **Insertion Sort** όπως στον SJF με δύο διαφορές. Η πρώτη και κύρια είναι ότι στο condition της σύγκρισης αντί να συγκρίνουμε τα **number** συγκρίνουμε τα **priority**. Η δεύτερη είναι ότι στην αρχή το **struct Work temp** είναι πίνακας με μέγεθος όσες

είναι οι διεργασίες. Στην ταξινόμηση χρησιμοποιούμε μόνο την πρώτη θέση σαν στον SJF που το **temp** δεν είναι πίνακας. Ο λόγος για τον οποίο κάναμε το **temp** πίνακα θα εξηγηθεί παρακάτω.

Έπειτα, εκτελούμε **enqueue()** για όλες τις διεργασίες για να τις βάλουμε στην ουρά αναμονής. Είναι καλό να σημειώσουμε πως ο **PRIO** λειτουργεί περίπου σαν συνδυασμό του **SJF** και του **RR**.

Δημιουργούμε μια μεταβλητή **bool** που γίνεται **true** όταν βρεθεί ίδιο **priority**,μια μεταβλητή **int** η οποία είναι η θέση του πρώτου στοιχείου που είχε ίδιο **priority** με το επόμενό του ή τα επόμενά του και δύο δείκτες τύπου **struct** Work που ο ένας δείχνει πάνω στο πίνακα **works** την διεργασία που βλέπουμε τώρα και ο άλλος στην ακριβώς επόμενη διεργασία. Φτιάχνουμε επίσης ένα **struct WorkQueue** με όνομα **temp_q** το οποίο θα χρησιμοποιηθεί για παρακάτω για τον **RR** που βρίσκεται μέσα στον **PRIO**. Κάνουμε αρχικοποίηση του **temp_q** και αρχίζουμε το χρονόμετρο για τον συνολικό χρόνο. Κάνουμε **dequeue()** ώστε το **first_work** να δείχνει στο πρώτο στοιχείο και μπαίνουμε σε μια λούπα με τόσες επαναλήψεις όσες είναι οι διεργασίες.

Αρχικοποιούμε το **pid** με **-1** που χρησιμοποιήται παρακάτω όπως στον **SJF** και φτιάχνουμε και μια μεταβλητή **int** την οποία θα εξηγήσουμε παρακάτω. Κάνουμε το **prev_work** να δείχνει στο πρώτο στοιχείο και προχωράμε την **first_work** να δείχνει στο ακριβώς επόμενο. Μετά ακολουθεί ενα **if** και ένα **else if** τα οποία θα εξηγήσουμε αργότερα. Για την ώρα πηγαίνυομε στο επόμενο **if** που ελέγχει αν το **priority** του στοιχείου που κοιτάμε είναι ίδιο με το **priority** του ακριβώς επόμενου. Αν είναι ίδια τότε κάνει την μεταβλητή **bool true** το οποίο δηλώνει ότι βρέθηκε στοιχείο με ίδιο **priority** και ξαναρχίζουμε από την αρχή της λούπας στην επόμενη όμως επανάληψη κατευθείαν.

Στην ουσία αυτό που θα γίνεται είναι όσο τα στοιχεία έχουν ίδιο **priority** απλά θα προχωράει μέχρι να βρούμε ότι υπάρχει κάπου διαφορετικό **priority** ή φτάσουμε στο τέλος του πίνακα,δηλαδή ο **first_work** να γίνει **NULL**.

Αν βρεθεί διαφορετικό **priority** τότε μπαίνει στο **if** που είναι μέσα στο **else**. Εκεί αναθέτουμε σε μια προσωρινή μεταβλητή που είχαμε δηλώσει παραπάνω την τιμή της θέσης του πρώτου στοιχείου που είχε ίδιο **priority** με όλα τα επόμενά του. Τρέχουμε μια άλλη λούπα στην οποία αναθέτουμε σε έναν προσωρινό πίνακα **temp**,ο ίδιος πίνακας με τον οποίο κάναμε την ταξινόμηση πιο πριν, όλες τις πληροφορίες των διεργασιών που είναι από την θέση με τιμή ίση με την τιμή της προσωρινής μεταβλητής μέχρι την θέση που δείχνει η **prev_work** πάνω στο πίνακα **works**.

Μετέπειτα, βάζουμε σε μια προσωρινή ουρά αναμονής τον προσωρινό πίνακα και εκτελούμε τον **RR** με παραμέτρους αυτά τα προσωρινά δεδομένα. Πρέπει να σημειώσουμε πως η μεταβλητή **start_of_same_priority** αντιστοιχεί ως η **index** παράμετρος του **RR** γιατί θέλουμε να αλλάζει τα περιεχόμενα του πίνακα **success** από αυτήν την αντίστοιχη θέση που βρίσκεται η πρώτη διεργασία με αυτό το **priority** πάνω στον πίνακα **works** και μετά. Μόλις τελιώσει ο **RR** ξανατρέχουμε μια λούπα για να κάνουμε την αντίστροφη διαδικασία της προπροηγούμενης λούπας πριν μπουμε στον **RR**.Κάνουμε την bool μεταβλητή **false** γιατί μπορεί να ξαναβρούμε διεργασίες με ίδιο **priority** και ενημερώνουμε την **start_of_same_priority** ώστε να είναι η θέση που θα κοιτάξουμε τώρα και μεταβαίνουμε στην επόμενη επανάληψη.

Τώρα, αν έχουμε φτάσει στο τέλος της ουράς και έχουμε βρεί ίδιο priority τότε εκτελείτε το **else if** με τα δύο conditions και κάνει ένα **goto** το οποίο μετακινεί την ροή του προγράμματος κατευθείαν για να εκτελεστεί ο **RR** και να τερματίσει μετά το πρόγραμμα.

Αν το πρόγραμμα δεν βρει ίδιο **priority** τότε μπαίνει μέσα στο **else** και βάζει το **start_of_same_priority** να είναι η επόμενη θέση σε περίπτωση που βρούμε ίδιο **priority** μετά και κάνουμε ότι είχαμε κάνει στον **SJF** με μόνη διαφορά πως μετράμε μέσα στον **PRIO** το **workload_prio** για όσες διεργασίες δεν έχουν ίδιο **priority** με κανένα άλλο.

Αν σε περίπτωση έχουμε φτάσει στο τέλος και δεν έχει βρεθεί ίδιο **priority** τότε εκτελούμε το τελευταίο **if** που μένει να εξηγήσουμε που έχει μέσα του ένα **goto** που μας μεταφέρει κατευθείαν στην εκτέλεση της διεργασίας.

1.3 Main

- Είναι σημαντικό να διαχωρίσουμε κάθε περίπτωση όπου ο χρήστης «καλεί» το εκτελέσιμο αρχείο '**scheduler**'. Αυτό το αρχείο μπορούμε να το καλέσουμε με 4 τρόπους:
  - ./scheduler BATCH reverse.txt
  - ./scheduler SJF reverse.txt
  - ./scheduler RR 1000 reverse.txt
  - ./scheduler PRIO 500 mixed.txt

  Ο πρώτος και ο δεύτερος τρόπος περιέχουν 3 **arguments** ενώ ο τρίτος και ο τέταρτος 4. Επισης, αν ο χρήστης χρησιμοποιήσει πλήθος **argument** διαφορετικό από τα παραπάνω τότε έχει κάνει λάθος. Είναι σημαντικό να ξεχωρίσουμε τις 2 περιπτώσεις που αναφέρθηκαν παραπάνω. Παρατηρούμε πως το όνομα του αλγορίθμου είναι σε όλες τις περιτώσεις στην θέση 1 δηλαδή, την θέση έπειτα την κλήση του αρχείου scheduler, οπότε θέτουμε για argv[1] την συμβολοσειρά που δηλώνει τον αλγόριθμο. Αν όμως είμαστε στην περίπτωση με τα 3 **arguments** το όνομα του αρχείου βρίσκεται στην θέση 2 δηλαδή, argv[2]. Ενώ στην περίπτωση με τα 4 arguments στην θέση 3, argv[3]. Τέλος στην τελευταία περίπτωση για θέση 2 αποθηκεύουμε το κβάντο, argv[2]. Παρακάτω είναι ο κώδικας που περιγράφει ακριβώς αυτό.

```
int main(int argc, char *argv[])
{

    // check if the number of arguments is correct
    if (argc != 3 && argc != 4)
    {
        printf("Invalid number of arguments. Usage: ./scheduler <algorithm> [quantum] <input_file> \n");
        return -1;
    }

    char *algorithm = argv[1];
    char *input_file;
```

```
if (argc == 3)
    {
        char *input_file = argv[2];
    }

    int quantum = 0;
    if (argc == 4)
    {

        input_file = argv[3];
        quantum = atoi(argv[2]);
    }
```

- Άνοιγμα και διάβασμα του αρχείου για τον προσδιορισμός του αριθμού των διεργασιών (**count**) και την αποθήκευση του πεδίου **command** και **priority** στο **struct processes** που θα χρησιμοποιήσουμε και αρχικοποίηση των υπόλοιπων πεδίων.

```
int count = 0;
  char row;
  FILE *file = fopen(input_file, "r");
  if (file == NULL)
  { printf("Could not open file\n");  return -1; }
  // metrame ton arithmo twn grammwn tou input_file
  for (row = getc(file); row != EOF; row = getc(file))
        if (row == '\n') // Increment count if this character is newline
            count++;
  rewind(file); // pigainoume ton file pointer stin arxi tou arxeiou
  char line[50];
  struct Work processes[count];
   char *token;
   int i = 0;
  int n; // apo to workn
  for (int i = 0; i < count; i++)
  {
     processes[i].time = 0;  processes[i].pid = -1;
    processes[i].status = -1; processes[i].exited = false;
  }
  char *token;
  int i = 0;
  int n; // workn
  while (fgets(line, 50, file) != NULL)
  {
     token = strtok(line, "\t");
     strcpy(processes[i].command, token);
     sscanf(token, "../work/work%d", &n); // diavazw ton arithmo tis diergasias
     processes[i].number = n;
     token = strtok(NULL, "\t");
     processes[i].priority = aprocesses[i].index = i;
     processes[i].time = 0;
     processes[i].pid = -1;
     processes[i].status = -1;
     processes[i].exited = false;toi(token);
     i++;
  }
  fclose(file);
```

- Πρέπει σύμφωνα με τον αλγόριθμο που διαβάστηκε από τα **arguments** να εκτελέσουμε και τον κάθε αλγόριθμο. Οπότε δημιουργούμε μία ουρά και την αρχικοποιούμε.

```
struct WorkQueue q;
init_queue(&q);
```

Έπειτα ακολουθεί η κλήση κάθε αλγορίθμου.

- **FCFS**

```
if (strcmp(algorithm, "BATCH") == 0)
    {
// populate queue
        for (int i = 0; i < count; i++)
        {
            enqueue(&q, &processes[i]);
        }
        FCFS(count, q, processes);
       printf("\n\n# scheduler %s %s\n\n", algorithm,
input_file);
    }
```

- **SJF**

  Για τον **SJF** όπως έχει προαναφερθεί χρειάστηκε να κάνουμε ταξινόμηση των διεργασιών με βάση την προτεραιότητας τους.

```
else if (strcmp(algorithm, "SJF") == 0)
    {
        SJF(count, q, processes);
        printf("\n\n# scheduler %s %s\n\n", algorithm, input_file);
    }
```

- **RR**

```
else if (strcmp(algorithm, "RR") == 0)
    {
        if (quantum == 0)
        {
            printf("Error: Quantum not specified for RR algorithm\n");
            return 1;
        }

        for (int i = 0; i < count; i++)
        {
          enqueue(&q, &processes[i]);
        }

        RR(quantum, &q, success, 0);
        workload_time = (end_rr.tv_sec - start_rr.tv_sec) + (end_rr.tv_usec - start_rr.tv_usec) /
1000000.0;
        printf("\n\n# scheduler %s %d %s\n\n", algorithm, quantum, input_file);
    }
```

- **PRIO**

```
else if (strcmp(algorithm,"PRIO")==0)
    {
        if (quantum==0)
        {
            printf("Error: Quantum not specified for PRIO algorithm\n");
            return 1;
        }

        PRIO(count,q,processes,quantum,success);
        workload_time = (end_prio.tv_sec - start_prio.tv_sec) + (end_prio.tv_usec -
start_prio.tv_usec) / 1000000.0;
        printf("\n\n# scheduler %s %d %s\n\n", algorithm, quantum, input_file);
    }
```

- Τέλος πρέπει και να εκτυπώσουμε τα αποτελέσματα. Οπότε αποθηκεύουμε σε μία μεταβλητή **workload** τον χρόνο 0.
  Για τις διεργασίες χρησιμοποιούμε έναν βρόγχο. Για την εκτύπωση του **workload** χρόνου για κάθε διεργασία, χρειάζεται να προσθέσουμε τον προηγούμενο **workload** χρόνο με τον πόσο χρειάστηκε κάθε εργασία για να ολοκληρωθεί.
  Τέλος εκτυπώνουμε και τον συνολικό χρόνο.

```
if(strcmp(algorithm, "BATCH") == 0 || strcmp(algorithm, "SJF") == 0)
 {
    double workload = 0;
    for (int i = 0; i < count; i++)
    {
       workload = workload + (processes[i].time);
       printf("Work: %d, Priority: %d, PID: %d, Elapsed Time: %.3lf, Workload Time: %.3lf\n",
          processes[i].number, processes[i].priority, processes[i].pid,
          processes[i].time, workload);
    }
  printf("\nWORKLOAD TIME: %.3lf\n\n", workload);
 }
 else if(strcmp(algorithm, "RR") == 0)
 {
    for (int i = 0; i < count; i++)
    {
       printf("Work: %d, Priority: %d, PID: %d, Elapsed Time: %.3lf, Workload Time: %.3lf\n",
          processes[success[i]].number, processes[success[i]].priority, processes[success[i]].pid,
          processes[success[i]].time,processes[success[i]].workload_rr);
    }
    printf("\nWORKLOAD TIME: %.3lf\n\n", workload_time);
 }
 else{
    for (int i = 0; i < count; i++)
    {
       printf("Work: %d, Priority: %d, PID: %d, Elapsed Time: %.3lf, Workload Time: %.3lf\n",
          processes[success[i]].number, processes[success[i]].priority, processes[success[i]].pid,
          processes[success[i]].time,processes[success[i]].workload_prio);
    }
    printf("\nWORKLOAD TIME: %.3lf\n\n", workload_time);
 } return 0;}
```

1.4 Αποτελέσματα εκτέλεσης του run.sh

Αφού κάνουμε make στον φάκελο work για να δημιουργηθούν τα εκτελέσιμα work1 έως work7 και compile το αρχείο scheduler.c, τρέχουμε το run.sh του οποίου τα αποτελέσματα είναι τα παρακάτω:

```
yes@yes-virtual-machine:~/Desktop/os_project2/scheduler$ ./run.sh
process 12726 begins
process 12726 ends
process 12730 begins
process 12730 ends
process 12731 begins
process 12731 ends
process 12732 begins
process 12732 ends
process 12733 begins
process 12733 ends


# scheduler BATCH homogeneous.txt

Work: 7, Priority: 7, PID: 12726, Elapsed Time: 8.670, Workload Time: 8.670
Work: 7, Priority: 7, PID: 12730, Elapsed Time: 10.403, Workload Time: 19.073
Work: 7, Priority: 7, PID: 12731, Elapsed Time: 8.997, Workload Time: 28.070
Work: 7, Priority: 7, PID: 12732, Elapsed Time: 8.475, Workload Time: 36.545
Work: 7, Priority: 7, PID: 12733, Elapsed Time: 8.769, Workload Time: 45.314

WORKLOAD TIME: 45.314

process 12735 begins
process 12735 ends
process 12736 begins
process 12736 ends
process 12737 begins
process 12737 ends
process 12738 begins
process 12738 ends
process 12739 begins
process 12739 ends


# scheduler SJF homogeneous.txt

Work: 7, Priority: 7, PID: 12735, Elapsed Time: 8.501, Workload Time: 8.501
Work: 7, Priority: 7, PID: 12736, Elapsed Time: 8.648, Workload Time: 17.149
Work: 7, Priority: 7, PID: 12737, Elapsed Time: 8.462, Workload Time: 25.611
Work: 7, Priority: 7, PID: 12738, Elapsed Time: 8.432, Workload Time: 34.043
Work: 7, Priority: 7, PID: 12739, Elapsed Time: 8.368, Workload Time: 42.410

WORKLOAD TIME: 42.410
```

```
process 12741 begins
process 12742 begins
process 12743 begins
process 12744 begins
process 12745 begins
process 12741 ends
process 12743 ends
process 12744 ends
process 12745 ends
process 12742 ends


# scheduler RR 1000 homogeneous.txt

Work: 7, Priority: 7, PID: 12741, Elapsed Time: 40.723, Workload Time: 40.724
Work: 7, Priority: 7, PID: 12743, Elapsed Time: 40.431, Workload Time: 42.436
Work: 7, Priority: 7, PID: 12744, Elapsed Time: 40.191, Workload Time: 43.197
Work: 7, Priority: 7, PID: 12745, Elapsed Time: 39.907, Workload Time: 43.914
Work: 7, Priority: 7, PID: 12742, Elapsed Time: 42.926, Workload Time: 43.927

WORKLOAD TIME: 43.927
```

```
process 12748 begins
process 12748 ends
process 12749 begins
process 12749 ends
process 12751 begins
process 12751 ends
process 12763 begins
process 12763 ends
process 12764 begins
process 12764 ends
process 12765 begins
process 12765 ends
process 12766 begins
process 12766 ends


# scheduler BATCH reverse.txt

Work: 7, Priority: 7, PID: 12748, Elapsed Time: 8.443, Workload Time: 8.443
Work: 6, Priority: 6, PID: 12749, Elapsed Time: 7.198, Workload Time: 15.641
Work: 5, Priority: 5, PID: 12751, Elapsed Time: 6.026, Workload Time: 21.667
Work: 4, Priority: 4, PID: 12763, Elapsed Time: 4.815, Workload Time: 26.483
Work: 3, Priority: 3, PID: 12764, Elapsed Time: 3.704, Workload Time: 30.187
Work: 2, Priority: 2, PID: 12765, Elapsed Time: 2.389, Workload Time: 32.576
Work: 1, Priority: 1, PID: 12766, Elapsed Time: 1.208, Workload Time: 33.783

WORKLOAD TIME: 33.783
```

```
process 12768 begins
process 12768 ends
process 12769 begins
process 12769 ends
process 12770 begins
process 12770 ends
process 12771 begins
process 12771 ends
process 12772 begins
process 12772 ends
process 12773 begins
process 12773 ends
process 12774 begins
process 12774 ends


# scheduler SJF reverse.txt

Work: 1, Priority: 1, PID: 12768, Elapsed Time: 1.201, Workload Time: 1.201
Work: 2, Priority: 2, PID: 12769, Elapsed Time: 2.407, Workload Time: 3.608
Work: 3, Priority: 3, PID: 12770, Elapsed Time: 3.611, Workload Time: 7.219
Work: 4, Priority: 4, PID: 12771, Elapsed Time: 4.975, Workload Time: 12.194
Work: 5, Priority: 5, PID: 12772, Elapsed Time: 6.077, Workload Time: 18.270
Work: 6, Priority: 6, PID: 12773, Elapsed Time: 7.222, Workload Time: 25.492
Work: 7, Priority: 7, PID: 12774, Elapsed Time: 8.409, Workload Time: 33.901

WORKLOAD TIME: 33.901
```

```
process 12776 begins
process 12777 begins
process 12778 begins
process 12779 begins
process 12780 begins
process 12781 begins
process 12782 begins
process 12782 ends
process 12781 ends
process 12780 ends
process 12779 ends
process 12778 ends
process 12777 ends
process 12776 ends


# scheduler RR 1000 reverse.txt

Work: 1, Priority: 1, PID: 12782, Elapsed Time: 7.210, Workload Time: 13.215
Work: 2, Priority: 2, PID: 12781, Elapsed Time: 13.630, Workload Time: 18.633
Work: 3, Priority: 3, PID: 12780, Elapsed Time: 19.260, Workload Time: 23.263
Work: 4, Priority: 4, PID: 12779, Elapsed Time: 27.308, Workload Time: 30.311
Work: 5, Priority: 5, PID: 12778, Elapsed Time: 30.527, Workload Time: 32.528
Work: 6, Priority: 6, PID: 12777, Elapsed Time: 33.059, Workload Time: 34.060
Work: 7, Priority: 7, PID: 12776, Elapsed Time: 34.741, Workload Time: 34.741

WORKLOAD TIME: 34.741

process 12784 begins
process 12785 begins
process 12784 ends
process 12785 ends
process 12786 begins
process 12787 begins
process 12786 ends
process 12787 ends


# scheduler PRIO 500 mixed.txt

Work: 4, Priority: 1, PID: 12784, Elapsed Time: 9.399, Workload Time: 9.399
Work: 6, Priority: 1, PID: 12785, Elapsed Time: 12.018, Workload Time: 12.519
Work: 5, Priority: 2, PID: 12786, Elapsed Time: 12.038, Workload Time: 24.558
Work: 7, Priority: 2, PID: 12787, Elapsed Time: 14.023, Workload Time: 27.042

WORKLOAD TIME: 27.042
```