ΚΩΤΣΗΣ ΣΤΑΘΗΣ 03115408
ΡΑΓΚΟΥΣΗΣ ΗΛΙΑΣ 03117897

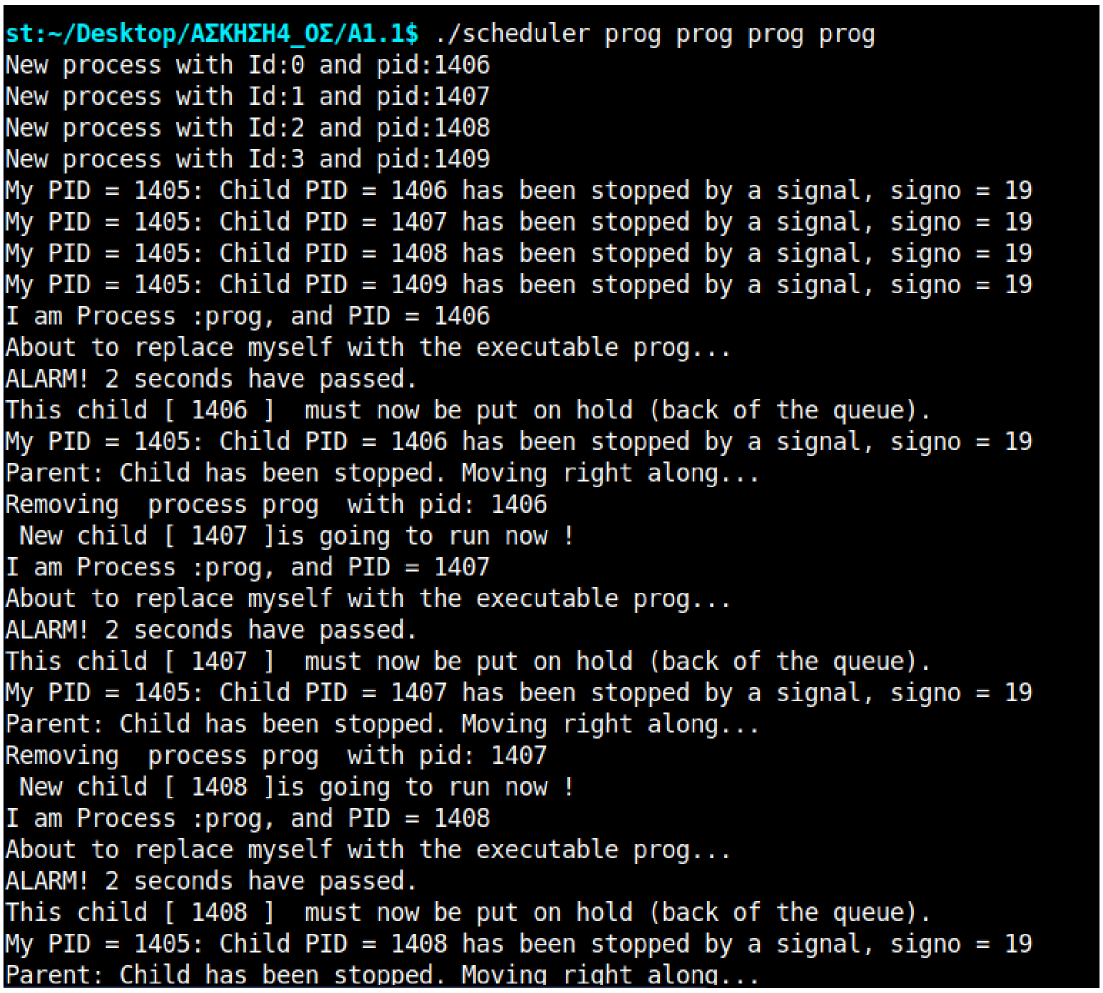
Λειτουργικά Συστήματα
6ο εξάμηνο, Ακαδημαϊκή περίοδος 2019-2020

Άσκηση 4: Χρονοδρομολόγηση

1.1 Υλοποίηση χρονοδρομολογητή κυκλικής επαναφοράς στο χώρο χρήστη

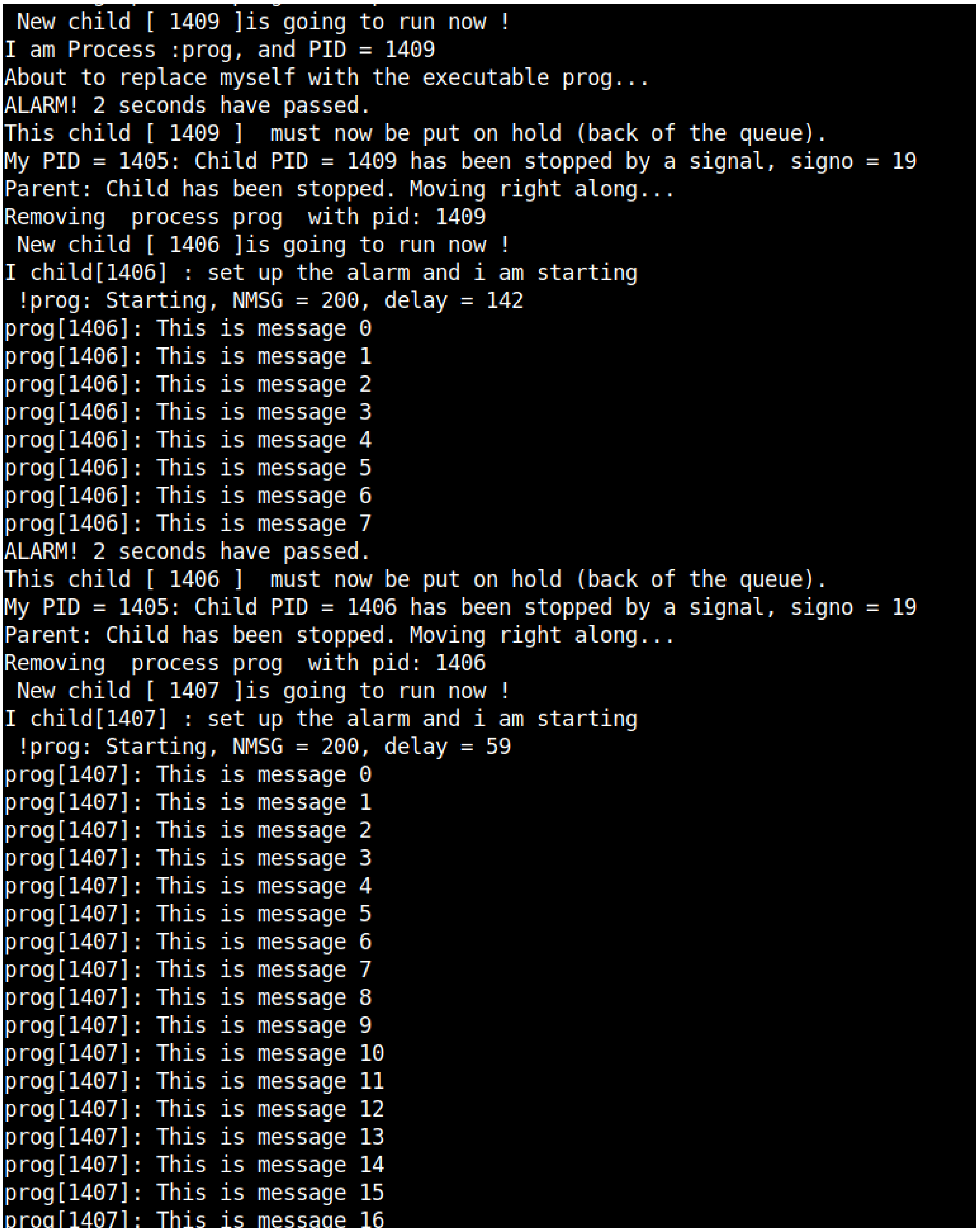
Τρέχουμε τον scheduler με : ./scheduler prog prog prog prog (π.χ)

Παρατηρούμε πρώτα ότι τα processes (prog) μπαίνουν στην ουρά που ορίσαμε και στην συνέχεια τα σταματάμε με SIGSTOP όλα εκτός από το 1ο το οποίο θα ξεκινήσει.Μόλις έρθει ALARM (κάθε 2 s) ή τελειώσει ,την θέση του θα πάρει το επόμενο process στην ουρά μέχρι να ολοκληρωθούν όλα.



```
st:~/Desktop/ΑΣΚΗΣΗ4_ΟΣ/A1.1$ ./scheduler prog prog prog prog
New process with Id:0 and pid:1406
New process with Id:1 and pid:1407
New process with Id:2 and pid:1408
New process with Id:3 and pid:1409
My PID = 1405: Child PID = 1406 has been stopped by a signal, signo = 19
My PID = 1405: Child PID = 1407 has been stopped by a signal, signo = 19
My PID = 1405: Child PID = 1408 has been stopped by a signal, signo = 19
My PID = 1405: Child PID = 1409 has been stopped by a signal, signo = 19
I am Process :prog, and PID = 1406
About to replace myself with the executable prog...
ALARM! 2 seconds have passed.
This child [ 1406 ]  must now be put on hold (back of the queue).
My PID = 1405: Child PID = 1406 has been stopped by a signal, signo = 19
Parent: Child has been stopped. Moving right along...
Removing  process prog  with pid: 1406
 New child [ 1407 ]is going to run now !
I am Process :prog, and PID = 1407
About to replace myself with the executable prog...
ALARM! 2 seconds have passed.
This child [ 1407 ]  must now be put on hold (back of the queue).
My PID = 1405: Child PID = 1407 has been stopped by a signal, signo = 19
Parent: Child has been stopped. Moving right along...
Removing  process prog  with pid: 1407
 New child [ 1408 ]is going to run now !
I am Process :prog, and PID = 1408
About to replace myself with the executable prog...
ALARM! 2 seconds have passed.
This child [ 1408 ]  must now be put on hold (back of the queue).
My PID = 1405: Child PID = 1408 has been stopped by a signal, signo = 19
Parent: Child has been stopped. Moving right along...
```

Στην συνέχεια βλέπουμε και την εναλλαγή των διεργασιών καθώς η μία σταματάει για να συνεχίσει η επόμενη στην ουρά.(dequeue και enqueue ο μηχανισμός για να αλλάξω process.)



```
 New child [ 1409 ]is going to run now !
I am Process :prog, and PID = 1409
About to replace myself with the executable prog...
ALARM! 2 seconds have passed.
This child [ 1409 ]  must now be put on hold (back of the queue).
My PID = 1405: Child PID = 1409 has been stopped by a signal, signo = 19
Parent: Child has been stopped. Moving right along...
Removing  process prog  with pid: 1409
 New child [ 1406 ]is going to run now !
I child[1406] : set up the alarm and i am starting
 !prog: Starting, NMSG = 200, delay = 142
prog[1406]: This is message 0
prog[1406]: This is message 1
prog[1406]: This is message 2
prog[1406]: This is message 3
prog[1406]: This is message 4
prog[1406]: This is message 5
prog[1406]: This is message 6
prog[1406]: This is message 7
ALARM! 2 seconds have passed.
This child [ 1406 ]  must now be put on hold (back of the queue).
My PID = 1405: Child PID = 1406 has been stopped by a signal, signo = 19
Parent: Child has been stopped. Moving right along...
Removing  process prog  with pid: 1406
 New child [ 1407 ]is going to run now !
I child[1407] : set up the alarm and i am starting
 !prog: Starting, NMSG = 200, delay = 59
prog[1407]: This is message 0
prog[1407]: This is message 1
prog[1407]: This is message 2
prog[1407]: This is message 3
prog[1407]: This is message 4
prog[1407]: This is message 5
prog[1407]: This is message 6
prog[1407]: This is message 7
prog[1407]: This is message 8
prog[1407]: This is message 9
prog[1407]: This is message 10
prog[1407]: This is message 11
prog[1407]: This is message 12
prog[1407]: This is message 13
prog[1407]: This is message 14
prog[1407]: This is message 15
prog[1407]: This is message 16
```

ΕΡΩΤΗΣΕΙΣ :

1)Αν έρθει SIGALARM ενώ τρέχει ο sigchld_handler ή SIGCHLD ενώ τρέχει ο alarm_handler το σήμα γίνεται mask και ο handler συνεχίζει μέχρι να ολοκληρωθεί.

Αυτό το πετυχαίνουμε με το
:install_signal_handlers() και συγκεκριμένα με το : sigaddset(&sigset,SIGCHLD);
sigaddset(&sigset,SIGALARM);
sa.sa_mask=sigset;

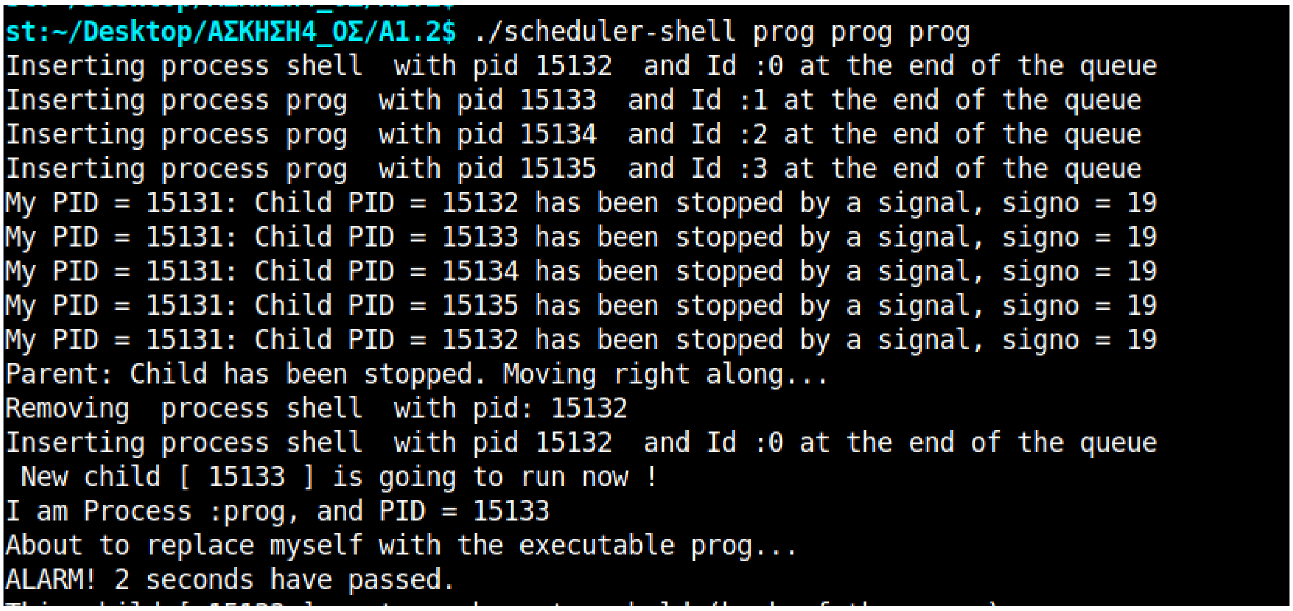
Μια πιο ρεαλιστική υλοποίηση στον χώρο πυρήνα θα έκανε handle τις διακοπές (interrupts) και όχι τα σήματα.

2)Όταν ο χρονοδρομολογητής λαμβάνει σήμα SIGCHLD , αυτό το σήμα αναφέρεται στη διεργασία-παιδί που είτε σταμάτησε λόγω SIGSTOP, είτε ολοκλήρωσε αυτό που είχε να κάνει και έκανε exit ή πέθανε ύστερα από SIGKILL. Αν λόγω εξωτερικού παράγοντα στείλουμε SIGKILL σε μια διεργασία τότε ο χρονοδρομολογητής τρέχει τον handler για τον SIGCHLD, αφαιρεί τη διεργασία από τη λίστα και συνεχίζει να χρονοδρομολογεί τις υπόλοιπες.Αν δίνουμε εμείς το SIGKILL μέσα σε συνάρτηση τότε είναι πιθανό απλώς να τερματίσει την διεργασία χωρίς απαραίτητα να έχω catch του σήματος.

3)Το SIGALARM από μόνο του δεν μας εξασφαλίζει ότι το SIGSTOP έχει γίνει catch από το process στο οποίο το στέλνουμε και ότι έχει σταματήσει.Έτσι το ελέγχουμε μέσα στον sigchld_handler για να είμαστε σίγουροι ότι το sigcont που έχω μέσα στον sigchld_handler ενεργοποιεί την σωστή -επόμενη διεργασία στην ουρά.

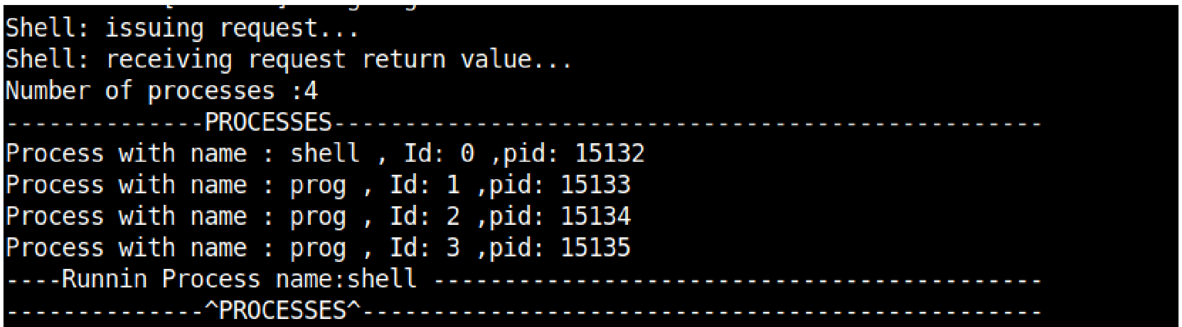
3.2
Άσκηση 1.2 Έλεγχος λειτουργίας χρονοδρομολογητή μέσω φλοιού

Εισάγουμε πλέον περισσότερες δυνατότητες στον scheduler-shell και σαν πρώτη στην ουρά εργασία το shell.(id =0) το οποίο χρονοδρομολογείται και αυτό.
Καταρχήν ξεκινάμε με τις παρακάτω διεργασίες:



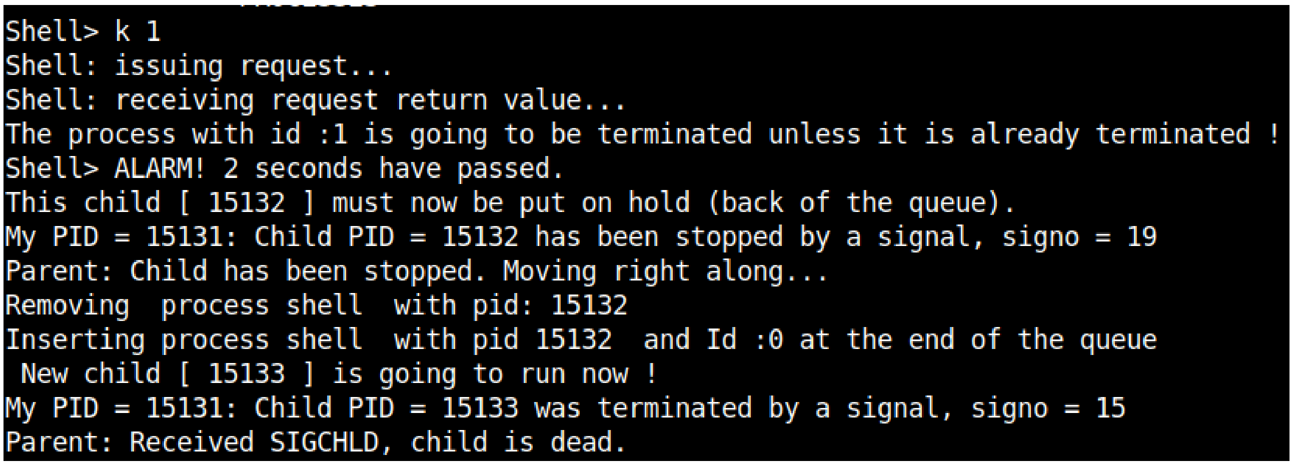
```
st:~/Desktop/ΑΣΚΗΣΗ4_ΟΣ/A1.2$ ./scheduler-shell prog prog prog
Inserting process shell  with pid 15132  and Id :0 at the end of the queue
Inserting process prog  with pid 15133  and Id :1 at the end of the queue
Inserting process prog  with pid 15134  and Id :2 at the end of the queue
Inserting process prog  with pid 15135  and Id :3 at the end of the queue
My PID = 15131: Child PID = 15132 has been stopped by a signal, signo = 19
My PID = 15131: Child PID = 15133 has been stopped by a signal, signo = 19
My PID = 15131: Child PID = 15134 has been stopped by a signal, signo = 19
My PID = 15131: Child PID = 15135 has been stopped by a signal, signo = 19
My PID = 15131: Child PID = 15132 has been stopped by a signal, signo = 19
Parent: Child has been stopped. Moving right along...
Removing  process shell  with pid: 15132
Inserting process shell  with pid 15132  and Id :0 at the end of the queue
 New child [ 15133 ] is going to run now !
I am Process :prog, and PID = 15133
About to replace myself with the executable prog...
ALARM! 2 seconds have passed.
```

Γράφοντας p στον φλοιό βλέπουμε τα processes μαζί με τα pid και τα Id τους.



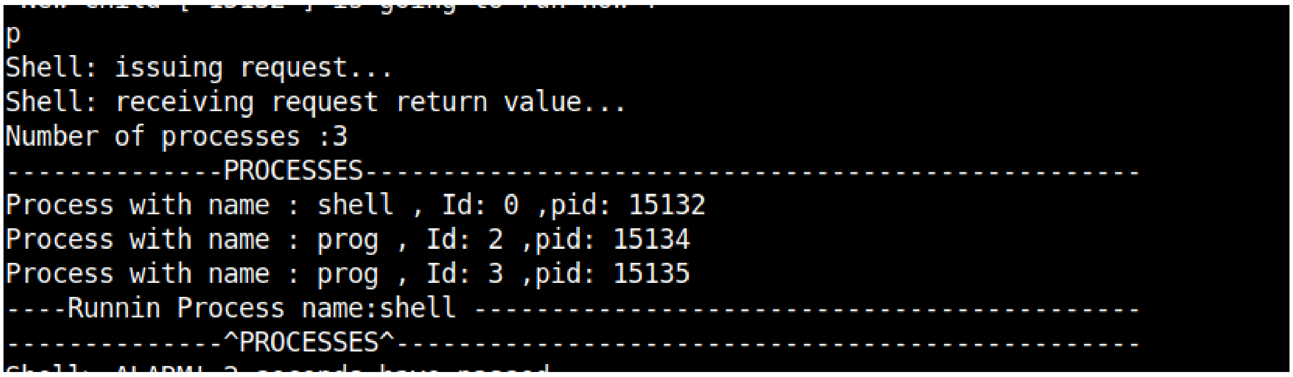
```
Shell: issuing request...
Shell: receiving request return value...
Number of processes :4
--------------PROCESSES------------------------------------------------
Process with name : shell , Id: 0 ,pid: 15132
Process with name : prog , Id: 1 ,pid: 15133
Process with name : prog , Id: 2 ,pid: 15134
Process with name : prog , Id: 3 ,pid: 15135
----Runnin Process name:shell -----------------------------------------
--------------^PROCESSES^---------------------------------------------
```

Στην συνέχεια με την k 1 ζητάμε τον τερματισμό (SIGTERM) της διεργασίας με id =1 που από ότι έχουμε δει αντιστοιχεί στο pid :15133



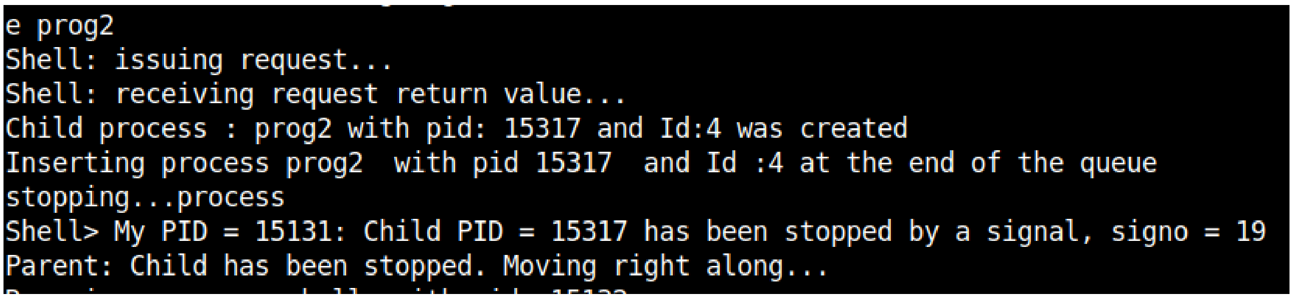
```
Shell> k 1
Shell: issuing request...
Shell: receiving request return value...
The process with id :1 is going to be terminated unless it is already terminated !
Shell> ALARM! 2 seconds have passed.
This child [ 15132 ] must now be put on hold (back of the queue).
My PID = 15131: Child PID = 15132 has been stopped by a signal, signo = 19
Parent: Child has been stopped. Moving right along...
Removing  process shell  with pid: 15132
Inserting process shell  with pid 15132  and Id :0 at the end of the queue
 New child [ 15133 ] is going to run now !
My PID = 15131: Child PID = 15133 was terminated by a signal, signo = 15
Parent: Received SIGCHLD, child is dead.
```

Οπότε παρατηρούμε ότι πλέον έχει αφαιρεθεί από την ουρά η διεργασία με id =1
Το βλέπουμε στο επόμενο p(print) .



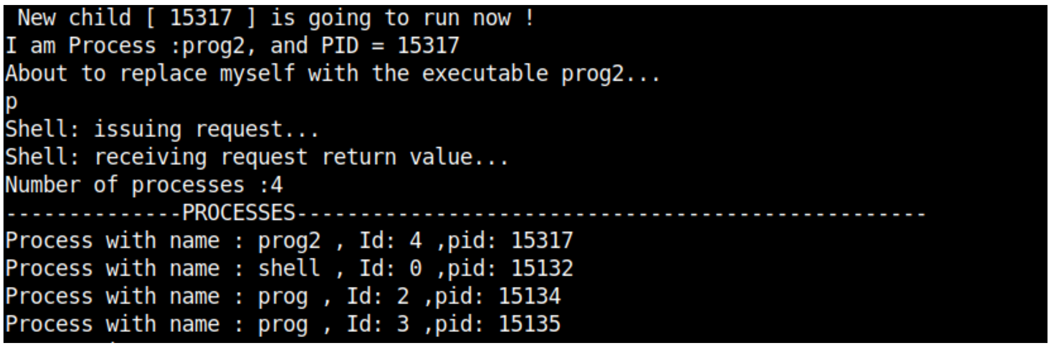
```
p
Shell: issuing request...
Shell: receiving request return value...
Number of processes :3
-------------PROCESSES-------------------------------------------------
Process with name : shell , Id: 0 ,pid: 15132
Process with name : prog , Id: 2 ,pid: 15134
Process with name : prog , Id: 3 ,pid: 15135
----Runnin Process name:shell -------------------------------------------
-------------^PROCESSES^------------------------------------------------
```

Γράφοντας e prog2 στην συνέχεια ζητάμε την δημιουργία μιας νέας διεργασίας η οποία θα ενσωματωθεί στην ουρά μας ,θα συμμετέχει στην χρονοδρομολόγηση και θα εκτελεί τις λειτουργίες που εκτελεί το executable_name (=prog2 εδώ) που δίνουμε και έχουμε στον κατάλογό μας.



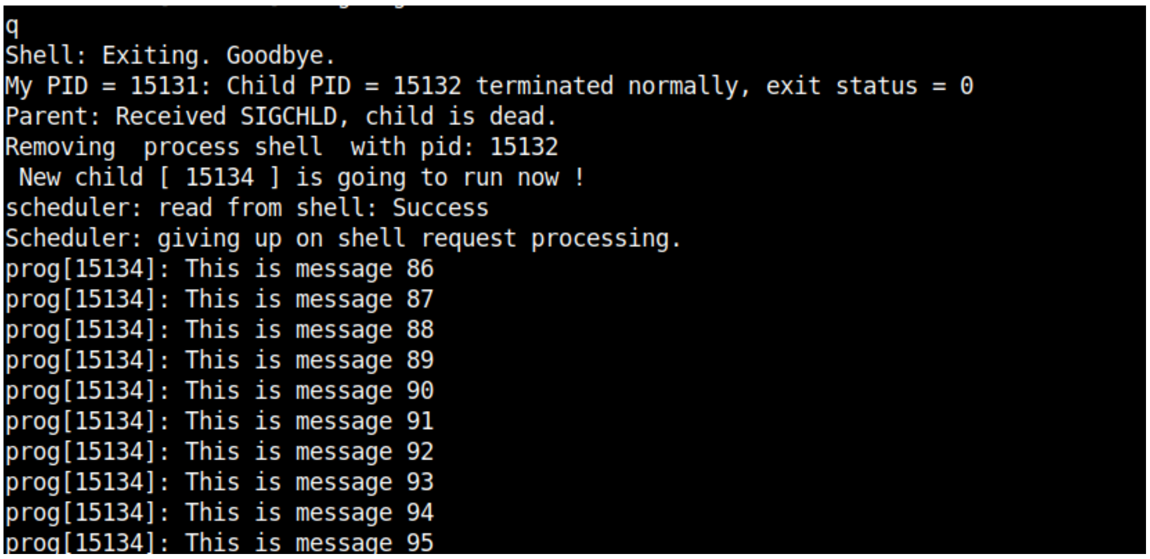
```
e prog2
Shell: issuing request...
Shell: receiving request return value...
Child process : prog2 with pid: 15317 and Id:4 was created
Inserting process prog2  with pid 15317  and Id :4 at the end of the queue
stopping...process
Shell> My PID = 15131: Child PID = 15317 has been stopped by a signal, signo = 19
Parent: Child has been stopped. Moving right along...
```

Ενώ στην επόμενη print βλέπουμε όλες τις τρέχουσες διεργασίες ,τις παλιές + την νέα.



```
 New child [ 15317 ] is going to run now !
I am Process :prog2, and PID = 15317
About to replace myself with the executable prog2...
p
Shell: issuing request...
Shell: receiving request return value...
Number of processes :4
-------------PROCESSES-------------------------------------------------
Process with name : prog2 , Id: 4 ,pid: 15317
Process with name : shell , Id: 0 ,pid: 15132
Process with name : prog , Id: 2 ,pid: 15134
Process with name : prog , Id: 3 ,pid: 15135
```

Τέλος αν γράψουμε q παρατηρούμε ότι ο φλοιός σταματάει την λειτουργία του ,κάτι που σημαίνει ότι στην συνέχεια όλες οι τρέχουσες διεργασίες συνεχίζουν να χρονοδρομολογούνται κυκλικά (ξεκινώντας από το pid :15134) χωρίς όμως πλέον την συμμετοχή του χρήστη μέσω του φλοιού.Μόλις τελειώσουν όλες τους θα τερματίσει και ο scheduler.



```
q
Shell: Exiting. Goodbye.
My PID = 15131: Child PID = 15132 terminated normally, exit status = 0
Parent: Received SIGCHLD, child is dead.
Removing  process shell  with pid: 15132
 New child [ 15134 ] is going to run now !
scheduler: read from shell: Success
Scheduler: giving up on shell request processing.
prog[15134]: This is message 86
prog[15134]: This is message 87
prog[15134]: This is message 88
prog[15134]: This is message 89
prog[15134]: This is message 90
prog[15134]: This is message 91
prog[15134]: This is message 92
prog[15134]: This is message 93
prog[15134]: This is message 94
prog[15134]: This is message 95
```

ΕΡΩΤΗΣΕΙΣ :

1)Ως τρέχουσα διεργασία θα εμφανίζεται ο φλοιός -shell γιατί πρακτικά εκείνος εκτελείται όταν γράφουμε p ,γίνεται στο δικό του κβάντο χρόνου άρα δεν γίνεται να εκτελείται κάποια άλλη διεργασία.

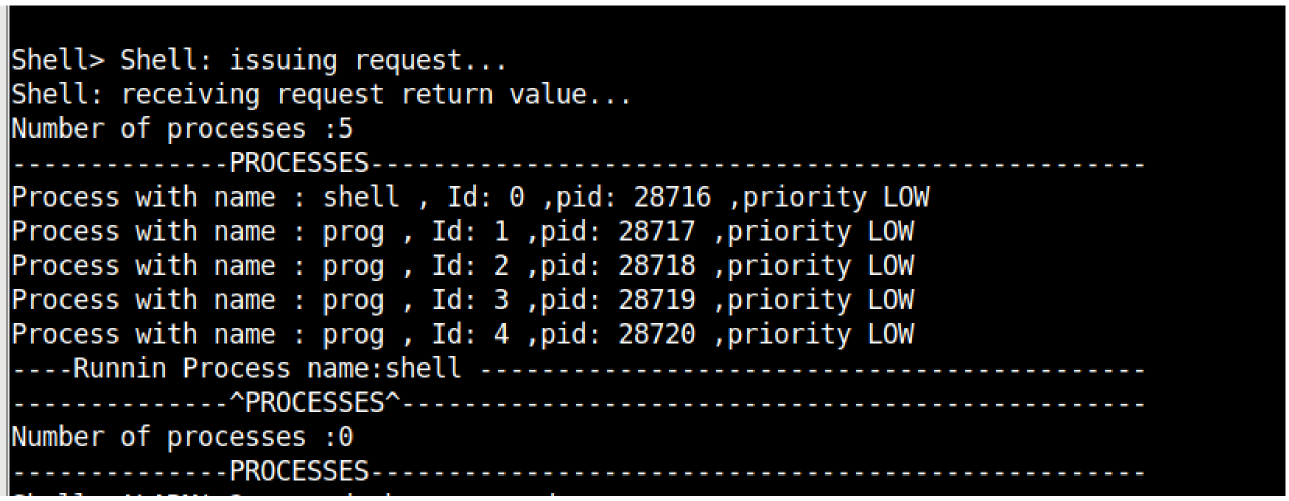
2)Απενεργοποιώντας τα σήματα όσο εξυπηρετείται ο φλοιός εξασφαλίζουμε ότι για παράδειγμα δε θα έρθει SIGALRM για να συνεχιστεί η επόμενη διεργασία την οποία εμείς μπορεί να έχουμε σκοτώσει. Προλαβαίνει έτσι δηλαδή ο χρονοδρομολογητής να ολοκληρώσει το αίτημα του shell χωρίς να τον διακόψει κάποιο σήμα.

1.3
Υλοποίηση προτεραιοτήτων στο χρονοδρομολογητή

Στο τελικό κομμάτι έκανα κάποιες υποθέσεις για την δομή του προγράμματος.
Α) Όλες οι διεργασίες ξεκινάνε με low και
Β)Η 1η αλλαγή low->high που θα γίνει από τον χρήστη σταματάει όλες τις low μαζί και τον φλοιό (εκτός και αν ο χρήστης κάνει αυτόν πρώτο high) και ο χρήστης πρέπει να περιμένει να τερματιστεί η high για να ξεκινήσουν πάλι οι low και να χει πρόσβαση και στον φλοιό.
Γ) Αν ο χρήστης θέλει να ορίσει μια ουρά από high που πρέπει να τελειώσουν πρώτα πριν από όλες τις low ,τότε θέτει πρώτα τον φλοιό high και μετά όποιες θέλει high.Έτσι θα χρονοδρομολογηθούν πρώτα όλες αυτές και όταν αδειάσει η high ουρά θα συνεχίσουν οι low processes.
Δ)Η υλοποίηση της p(print) τώρα βγάζει και τις 2 ουρές
->low processes (pt) ,
->high processes(highQ)

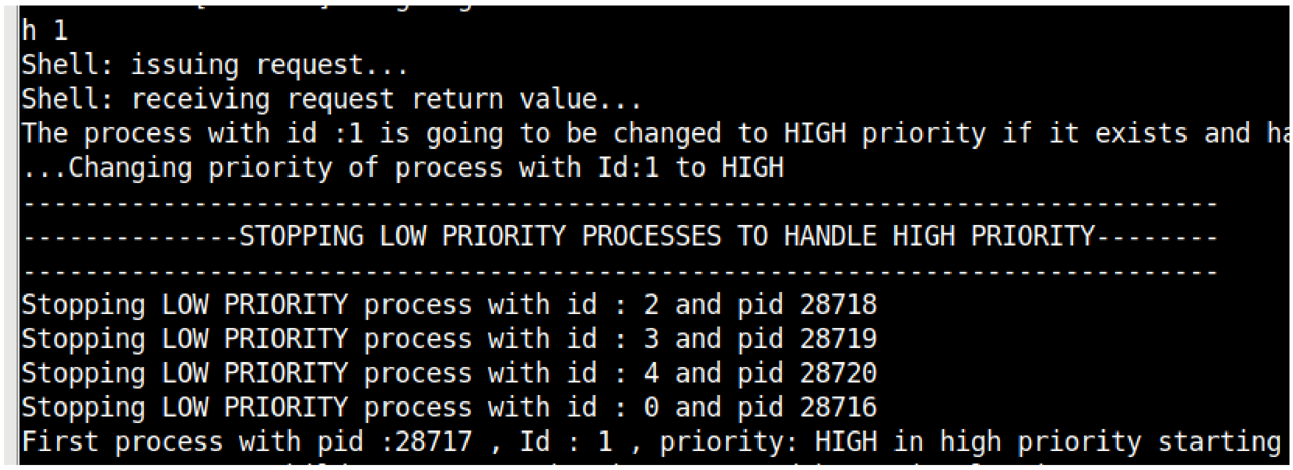
Έτσι για ένα παράδειγμα εκτέλεσης έχουμε :

Ξεκινάμε με τις εξής διεργασίες (p)



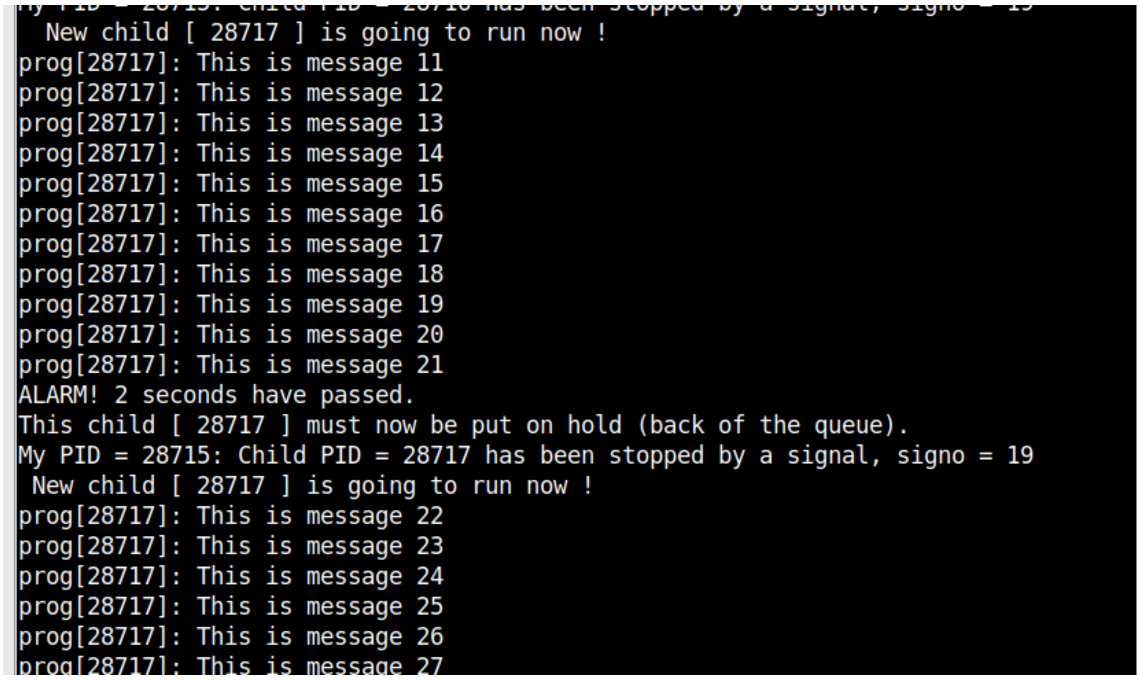
```
Shell> Shell: issuing request...
Shell: receiving request return value...
Number of processes :5
--------------PROCESSES---------------------------------------------------
Process with name : shell , Id: 0 ,pid: 28716 ,priority LOW
Process with name : prog , Id: 1 ,pid: 28717 ,priority LOW
Process with name : prog , Id: 2 ,pid: 28718 ,priority LOW
Process with name : prog , Id: 3 ,pid: 28719 ,priority LOW
Process with name : prog , Id: 4 ,pid: 28720 ,priority LOW
----Runnin Process name:shell ---------------------------------------------
--------------^PROCESSES^-------------------------------------------------
Number of processes :0
--------------PROCESSES---------------------------------------------------
```

Στην συνέχεια αλλάζουμε την διεργασία με Id =1 σε High priority οπότε εκτελείται πλέον μόνο αυτή μέχρι να ολοκληρωθεί



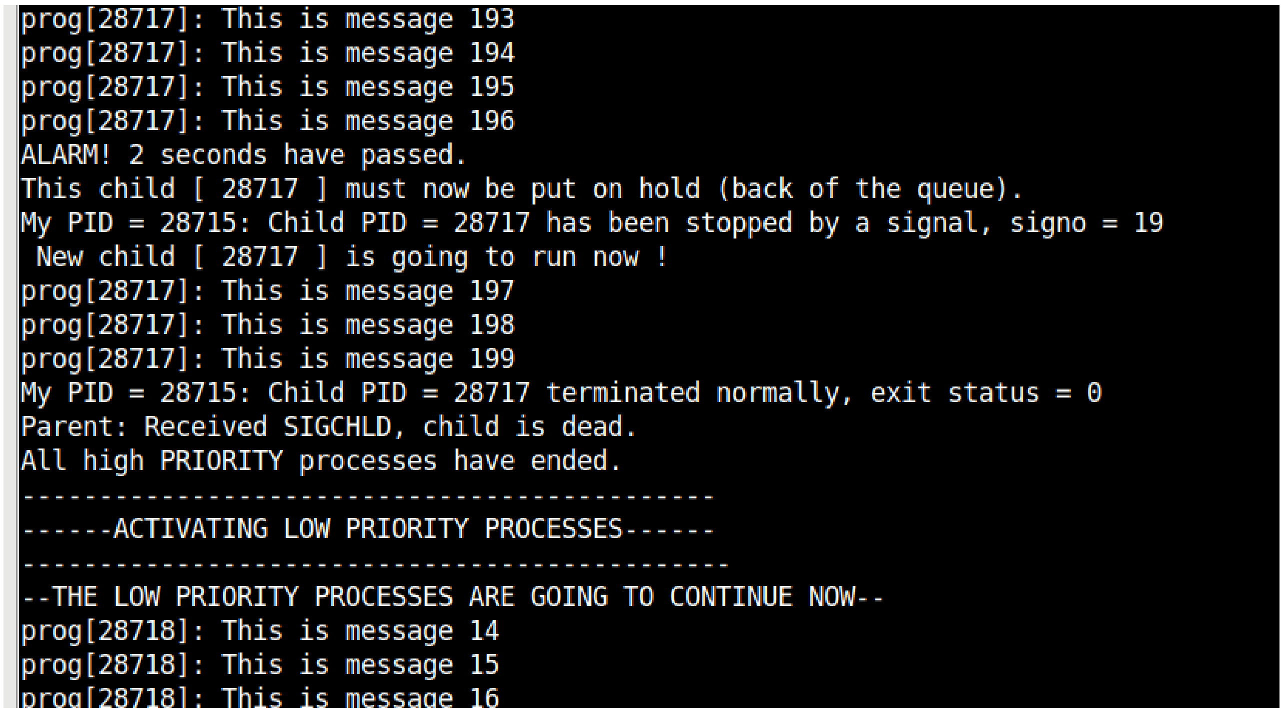
```
h 1
Shell: issuing request...
Shell: receiving request return value...
The process with id :1 is going to be changed to HIGH priority if it exists and ha
...Changing priority of process with Id:1 to HIGH
----------------------------------------------------------------------------
--------------STOPPING LOW PRIORITY PROCESSES TO HANDLE HIGH PRIORITY--------
----------------------------------------------------------------------------
Stopping LOW PRIORITY process with id : 2 and pid 28718
Stopping LOW PRIORITY process with id : 3 and pid 28719
Stopping LOW PRIORITY process with id : 4 and pid 28720
Stopping LOW PRIORITY process with id : 0 and pid 28716
First process with pid :28717 , Id : 1 , priority: HIGH in high priority starting
```

πάνω η αλλαγή και κάτω η εκτέλεση πλέον μόνο της pid:28717 , id=1



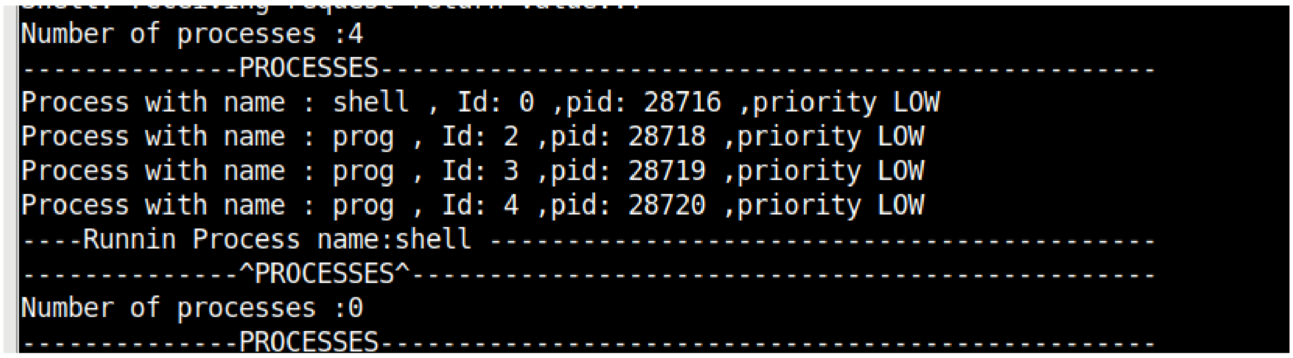
```
My PID = 28715: Child PID = 28716 has been stopped by a signal, signo = 19
  New child [ 28717 ] is going to run now !
prog[28717]: This is message 11
prog[28717]: This is message 12
prog[28717]: This is message 13
prog[28717]: This is message 14
prog[28717]: This is message 15
prog[28717]: This is message 16
prog[28717]: This is message 17
prog[28717]: This is message 18
prog[28717]: This is message 19
prog[28717]: This is message 20
prog[28717]: This is message 21
ALARM! 2 seconds have passed.
This child [ 28717 ] must now be put on hold (back of the queue).
My PID = 28715: Child PID = 28717 has been stopped by a signal, signo = 19
 New child [ 28717 ] is going to run now !
prog[28717]: This is message 22
prog[28717]: This is message 23
prog[28717]: This is message 24
prog[28717]: This is message 25
prog[28717]: This is message 26
prog[28717]: This is message 27
```

Μόλις ολοκληρωθεί λοιπόν βλέπουμε ότι συνεχίζουν οι low priority processes :



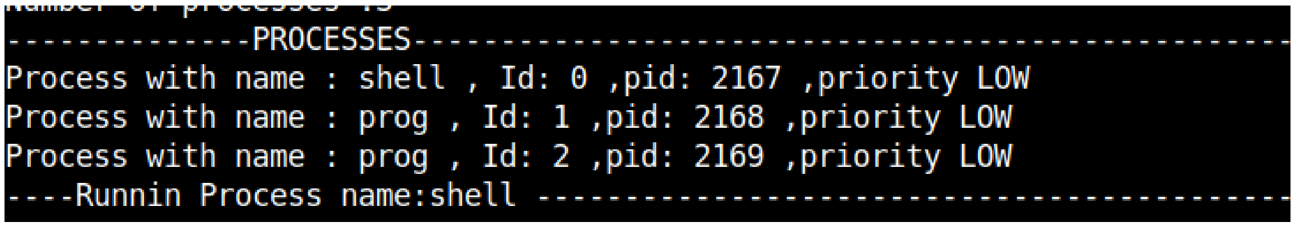
```
prog[28717]: This is message 193
prog[28717]: This is message 194
prog[28717]: This is message 195
prog[28717]: This is message 196
ALARM! 2 seconds have passed.
This child [ 28717 ] must now be put on hold (back of the queue).
My PID = 28715: Child PID = 28717 has been stopped by a signal, signo = 19
 New child [ 28717 ] is going to run now !
prog[28717]: This is message 197
prog[28717]: This is message 198
prog[28717]: This is message 199
My PID = 28715: Child PID = 28717 terminated normally, exit status = 0
Parent: Received SIGCHLD, child is dead.
All high PRIORITY processes have ended.
---------------------------------------------
------ACTIVATING LOW PRIORITY PROCESSES------
---------------------------------------------
--THE LOW PRIORITY PROCESSES ARE GOING TO CONTINUE NOW--
prog[28718]: This is message 14
prog[28718]: This is message 15
prog[28718]: This is message 16
```

και με print (p) βλέπουμε όσες processes έχουν μείνει στην low ουρά και εκτελούνται.



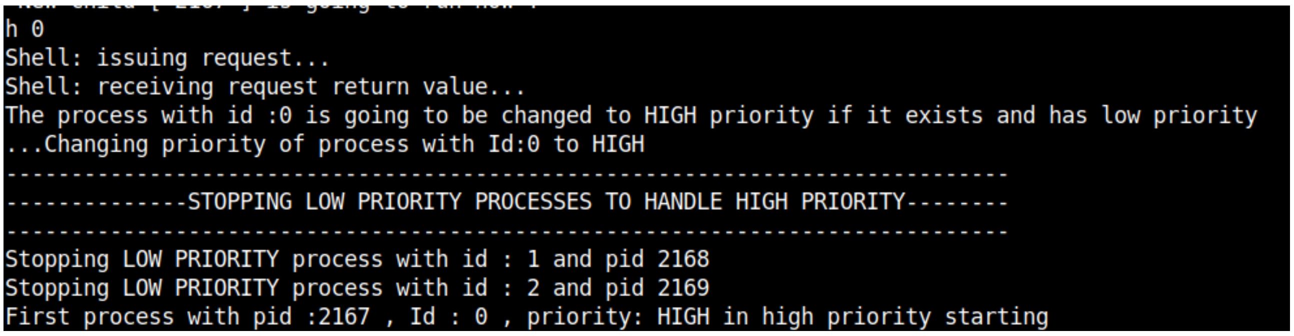
```
Number of processes :4
--------------PROCESSES-----------------------------------------------
Process with name : shell , Id: 0 ,pid: 28716 ,priority LOW
Process with name : prog , Id: 2 ,pid: 28718 ,priority LOW
Process with name : prog , Id: 3 ,pid: 28719 ,priority LOW
Process with name : prog , Id: 4 ,pid: 28720 ,priority LOW
----Runnin Process name:shell -----------------------------------------
--------------^PROCESSES^---------------------------------------------
Number of processes :0
--------------PROCESSES-----------------------------------------------
```

Μια 2η εναλλακτική εκτέλεση είναι το σενάριο Γ- Δ που αναφέρθηκε παραπάνω
Έστω λοιπόν ο χρήστης κάνει High τον φλοιό και μετά από αυτόν όποια άλλη διεργασία θέλει:Έχουμε αρχικά τις διεργασίες :



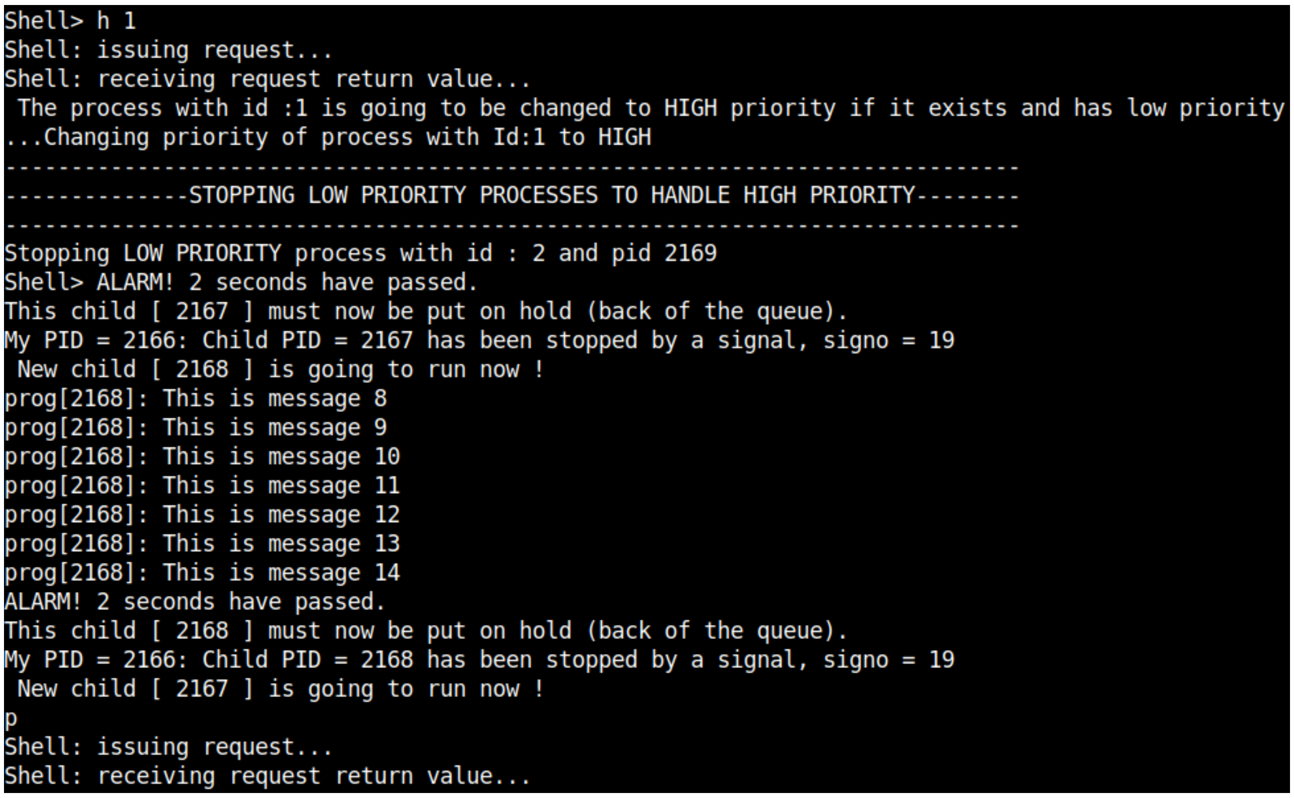
```
--------------PROCESSES-----------------------------------------------
Process with name : shell , Id: 0 ,pid: 2167 ,priority LOW
Process with name : prog , Id: 1 ,pid: 2168 ,priority LOW
Process with name : prog , Id: 2 ,pid: 2169 ,priority LOW
----Runnin Process name:shell -----------------------------------------
```

Στην συνέχεια ο φλοιός γίνεται high



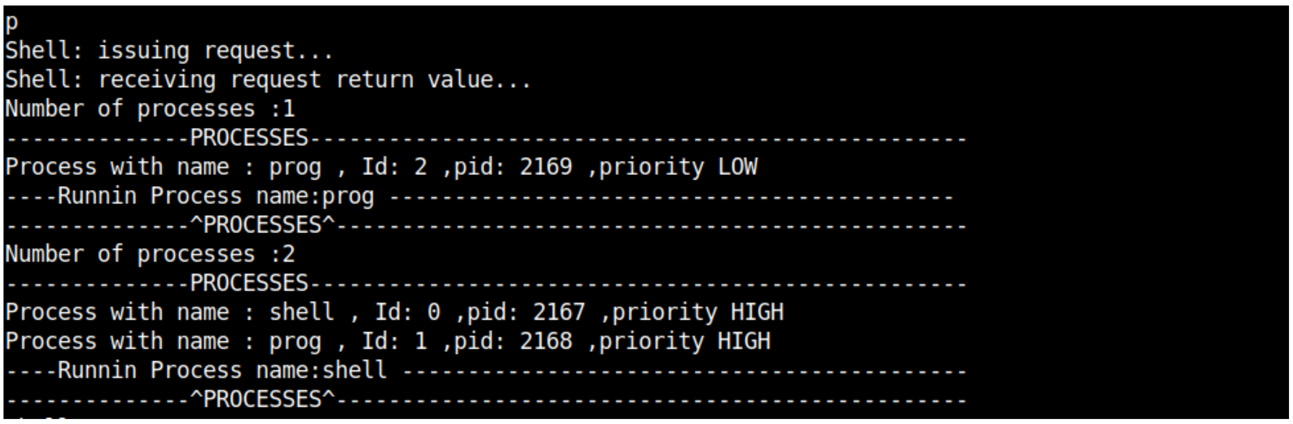
```
h 0
Shell: issuing request...
Shell: receiving request return value...
The process with id :0 is going to be changed to HIGH priority if it exists and has low priority
...Changing priority of process with Id:0 to HIGH
-------------------------------------------------------------------------
-------------STOPPING LOW PRIORITY PROCESSES TO HANDLE HIGH PRIORITY--------
-------------------------------------------------------------------------
Stopping LOW PRIORITY process with id : 1 and pid 2168
Stopping LOW PRIORITY process with id : 2 and pid 2169
First process with pid :2167 , Id : 0 , priority: HIGH in high priority starting
```

και εν συνεχεία από τον φλοιό ο χρήστης κάνει high την διεργασία με id=1 οπότε πλέον εναλλάσονται αυτές οι high .



```
Shell> h 1
Shell: issuing request...
Shell: receiving request return value...
 The process with id :1 is going to be changed to HIGH priority if it exists and has low priority
...Changing priority of process with Id:1 to HIGH
-----------------------------------------------------------------------------
--------------STOPPING LOW PRIORITY PROCESSES TO HANDLE HIGH PRIORITY--------
-----------------------------------------------------------------------------
Stopping LOW PRIORITY process with id : 2 and pid 2169
Shell> ALARM! 2 seconds have passed.
This child [ 2167 ] must now be put on hold (back of the queue).
My PID = 2166: Child PID = 2167 has been stopped by a signal, signo = 19
 New child [ 2168 ] is going to run now !
prog[2168]: This is message 8
prog[2168]: This is message 9
prog[2168]: This is message 10
prog[2168]: This is message 11
prog[2168]: This is message 12
prog[2168]: This is message 13
prog[2168]: This is message 14
ALARM! 2 seconds have passed.
This child [ 2168 ] must now be put on hold (back of the queue).
My PID = 2166: Child PID = 2168 has been stopped by a signal, signo = 19
 New child [ 2167 ] is going to run now !
p
Shell: issuing request...
Shell: receiving request return value...
```

Τέλος με p βλέπουμε και τις high priority και low priority στις 2 λίστες.



```
p
Shell: issuing request...
Shell: receiving request return value...
Number of processes :1
--------------PROCESSES---------------------------------------------------
Process with name : prog , Id: 2 ,pid: 2169 ,priority LOW
----Runnin Process name:prog --------------------------------------------
--------------^PROCESSES^-------------------------------------------------
Number of processes :2
--------------PROCESSES---------------------------------------------------
Process with name : shell , Id: 0 ,pid: 2167 ,priority HIGH
Process with name : prog , Id: 1 ,pid: 2168 ,priority HIGH
----Runnin Process name:shell --------------------------------------------
--------------^PROCESSES^-------------------------------------------------
```

ΕΡΩΤΗΣΕΙΣ

1)Λιμοκτονία δημιουργείται εύκολα ως εξής: Αν δηλαδή ο χρήστης απλώς έχει θέσει μια διεργασία Α (μη διαδραστική όπως ο φλοιός) σε high ή πολλές από αυτές και κάθε φορά οι low περιμένουν ενώ ταυτόχρονα τις νέες διεργασίες (από create (e)) τις κάνει high τότε η ουρά των high θα αυξάνεται και θα μειώνεται γύρω από κάποια τιμή αλλα δεν θα μηδενίζεται με αποτέλεσμα οι low να μην ενεργοποποιηθούν ποτέ.

ΚΩΔΙΚΕΣ :

ΑΣΚ 1.1

```
#include <assert.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <sys/types.h>
#include <sys/wait.h>

#include "proc-common.h"
#include "request.h"

/* Compile-time parameters. */
#define SCHED_TQ_SEC 2  /* time quantum */
#define TASK_NAME_SZ 10 /* maximum size for a task's name */
#define SLEEP_SEC 1
#define ALARM_SEC 2

typedef struct ProcessList {
  pid_t procid;
  struct ProcessList *next;
  char procname[10];
  int TaskId;
} ProcessList;

struct queue {
  ProcessList *items; // array to store queue elements
  int maxsize;        // maximum capacity of the queue
  int front;          // front points to front element in the queue (if any)
  int rear;           // rear points to last element in the queue
  int size;           // current capacity of the queue
};

// Utility function to initialize queue
struct queue *newProcessQueue(int size) {
  struct queue *pt = NULL;
  pt = (struct queue *)malloc(sizeof(struct queue));

  pt->items = (ProcessList *)malloc(size * sizeof(ProcessList));
  pt->maxsize = size;
  pt->front = 0;
  pt->rear = -1;
  pt->size = 0;

  return pt;
}

// Utility function to return the size of the queue
```

```
int size(struct queue *pt) { return pt->size; }

// Utility function to check if the queue is empty or not
int isEmpty(struct queue *pt) { return !size(pt); }

// Utility function to return front element in queue
ProcessList front(struct queue *pt) {
  if (isEmpty(pt)) {
    printf("UnderFlow\nProgram Terminated\n");
    exit(EXIT_FAILURE);
  }

  return pt->items[pt->front];
}

/* function to enqueue  process */
void enqueue(struct queue *pt, ProcessList x) {
  if (size(pt) == pt->maxsize) {
    printf("OverFlow\nProgram Terminated\n");
    exit(EXIT_FAILURE);
  }

  // printf("Inserting process %s  with pid %ld and Id %d at the end of the
  // queue \n", x.procname,x.procid,x.TaskId);

  pt->rear = (pt->rear + 1) % pt->maxsize; // circular queue
  pt->items[pt->rear] = x;
  pt->size++;

  // printf("front = %d, rear = %d\n", pt->front, pt->rear);
}

/*function to dequeue process */
void dequeue(struct queue *pt) {
  if (isEmpty(pt)) // front == rear
  {
    printf("UnderFlow\nProgram Terminated\n");
    exit(EXIT_FAILURE);
  }

  printf("Removing  process %s  with pid: %ld \n", (front(pt)).procname,
         (front(pt)).procid);

  pt->front = (pt->front + 1) % pt->maxsize; // circular queue
  pt->size--;

  // printf("front = %d, rear = %d\n", pt->front, pt->rear);
}

/* finished Queue implementation  */

/*handlers and main() */
```

```
/*initializing global pointers */
pid_t cur_pid = 0;
ProcessList *current;
struct queue *pt = NULL;
ProcessList *process = NULL;
pid_t *pid = NULL;

/*
 * SIGALRM handler
 */
static void sigalrm_handler(int signum) {
  if (signum != SIGALRM) {
    fprintf(stderr, "Internal error: Called for signum %d, not SIGALRM\n",
            signum);
    exit(1);
  }

  printf("ALARM! %d seconds have passed.\n", ALARM_SEC);

  // sigstop the process and put at the end of the queue
  printf("This child [ %ld ]  must now be put on hold (back of the queue).\n",
         cur_pid);
  kill(cur_pid, SIGSTOP);
}

/*
 * SIGCHLD handler
 */
static void sigchld_handler(int signum) {

  pid_t p;
  int status;

  if (signum != SIGCHLD) {
    fprintf(stderr, "Internal error: Called for signum %d, not SIGCHLD\n",
            signum);
    exit(1);
  }

  /*
   * Something has happened to one of the children.
   * We use waitpid() with the WUNTRACED flag, instead of wait(), because
   * SIGCHLD may have been received for a stopped, not dead child.
   *
   * A single SIGCHLD may be received if many processes die at the same time.
   * We use waitpid() with the WNOHANG flag in a loop, to make sure all
   * children are taken care of before leaving the handler.
   */

  for (;;) {
    p = waitpid(-1, &status, WUNTRACED | WNOHANG);
```

```
if (p < 0) {
  perror("waitpid");
  exit(1);
}
if (p == 0)
  break;

explain_wait_status(p, status);

if (WIFEXITED(status) || WIFSIGNALED(status)) {
  /* A child has died */
  printf("Parent: Received SIGCHLD, child is dead.\n");

  // remove from queue the child that finished
  // if it was the last one --> all processes finished
  if (!isEmpty(pt)) {
    dequeue(pt);
    // printf(" queue size %d\n: ",size(pt));
  }

  // this is just a guarding else -->should not go here
  else {
    printf("All processes have ended !Exiting scheduler...\n");
    exit(1);
  }

  // if after removing the last process the queue is empty then we are done
  // !
  if (isEmpty(pt)) {
    printf("All processes have ended !Exiting scheduler...\n");
    exit(1);
  }

  printf(" New child [ %ld ] is going to run now !\n", front(pt).procid);

  // wake up the next process
  cur_pid = front(pt).procid;
  kill(cur_pid, SIGCONT);
  current = &pt->items[pt->front];

  /* Setup the alarm again */
  if (alarm(SCHED_TQ_SEC) < 0) {
    perror("alarm");
    exit(1);
  }
}

if (WIFSTOPPED(status)) {
  /* A child has stopped due to SIGSTOP/SIGTSTP, etc... */
  printf("Parent: Child has been stopped. Moving right along...\n");

  // remove the child and reattach it at the end of the queue
```

```
      dequeue(pt);
      enqueue(pt, *current);

      printf(" New child [ %ld ]is going to run now !\n", front(pt).procid);

      cur_pid = front(pt).procid;
      kill(cur_pid, SIGCONT);
      current = &pt->items[pt->front];

      /* Setup the alarm again */
      if (alarm(ALARM_SEC) < 0) {
        perror("alarm");
        exit(1);
      }
    }
  }
}

/* Install two signal handlers.
 * One for SIGCHLD, one for SIGALRM.
 * Make sure both signals are masked when one of them is running.
 */
static void install_signal_handlers(void) {
  sigset_t sigset;
  struct sigaction sa;

  sa.sa_handler = sigchld_handler;
  sa.sa_flags = SA_RESTART;
  sigemptyset(&sigset);
  sigaddset(&sigset, SIGCHLD);
  sigaddset(&sigset, SIGALRM);

  sa.sa_mask = sigset;

  // handling sigchld
  if (sigaction(SIGCHLD, &sa, NULL) < 0) {
    perror("sigaction: sigchld");
    exit(1);
  }

  // handling sigalarm
  sa.sa_handler = sigalrm_handler;

  if (sigaction(SIGALRM, &sa, NULL) < 0) {
    perror("sigaction: sigalrm");
    exit(1);
  }

  /*
   * Ignore SIGPIPE, so that write()s to pipes
   * with no reader do not result in us being killed,
   * and write() returns EPIPE instead.
   */
  if (signal(SIGPIPE, SIG_IGN) < 0) {
    perror("signal: sigpipe");
    exit(1);
  }
}

/* all prog processes do this process */
/*  prog - child process */
void doChild(char *procname) {

  char executable[] = "prog";
  char *newargv[] = {executable, NULL, NULL, NULL}; //& exexutable[0]
  char *newenviron[] = {NULL};

  printf("I am Process :%s, and PID = %ld\n", procname, (long)getpid());
  printf("About to replace myself with the executable %s...\n", executable);
  sleep(2);

  execve(executable, newargv, newenviron);

  /* execve() only returns on error */
  perror("execve");
  exit(1);
}

int main(int argc, char *argv[]) {
  int nproc;
  int i = 0;
  /*
   * For each of argv[1] to argv[argc - 1],
   * create a new child process, add it to the process list.
   */

  /* initialize the structures ,pointers */
  process = (ProcessList *)realloc(process, (argc + 1) * sizeof(ProcessList));
  pid = (pid_t *)realloc(pid, (argc + 1) * sizeof(pid_t));
  pt = newProcessQueue(argc + 1);

  /* create all the processes ,freeze all but one */
  for (i = 1; i <= argc - 1; i++) {

    pid[i - 1] = fork();

    if (pid[i - 1] == 0) {
      doChild(argv[i]);
    }

    if (pid[i - 1] < 0) {

      perror("error forking child ");
      exit(1);
    }

    process[i - 1].procid = pid[i - 1];
    strcpy(process[i - 1].procname, argv[i]);
    enqueue(pt, process[i - 1]);
    process[i - 1].TaskId = i - 1;
    printf("New process with Id:%d and pid:%ld\n", process[i - 1].TaskId,
           process[i - 1].procid);
    kill(process[i - 1].procid, SIGSTOP);
  }

  // start the first process //

  nproc = argc - 1; /* number of proccesses goes here */

  // SET UP THE ALARM
  // alarm(SCHED_TQ_SEC);

  /* Wait for all children to raise SIGSTOP before exec()ing. */
  wait_for_ready_children(nproc);

  /* Install SIGALRM and SIGCHLD handlers. */
  install_signal_handlers();

  if (nproc == 0) {
    fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
    exit(1);
  }

  /* start the first process */
  cur_pid = front(pt).procid;
  current = &pt->items[pt->front];
  kill(cur_pid, SIGCONT);
  if (alarm(SCHED_TQ_SEC) < 0) {
    perror("alarm");
    exit(1);
  }

  /* loop forever  until we exit from inside a signal handler. */
  while (pause()) {
    //  alarm (SCHED_TQ_SEC);
  }

  /* Unreachable */
  fprintf(stderr, "Internal error: Reached unreachable point\n");

  free(process);
  free(pid);
  return 1;
}
```

ΑΣΚ1.2

```
#include <assert.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <stdbool.h>
#include <sys/types.h>
#include <sys/wait.h>

#include "proc-common.h"
#include "request.h"

/* Compile-time parameters. */
#define SCHED_TQ_SEC 10              /* time quantum */
#define TASK_NAME_SZ 60              /* maximum size for a task's name */
#define SHELL_EXECUTABLE_NAME "shell" /* executable for shell */
#define SLEEP_SEC 1
#define ALARM_SEC 2

/* The function of the prog processses-children */
void doChild(char *procname) {
  char *newargv[] = {procname, NULL, NULL, NULL}; //& exexutable[0]
  char *newenviron[] = {NULL};

  printf("I am Process :%s, and PID = %ld\n", procname, (long)getpid());
  printf("About to replace myself with the executable %s...\n", procname);
  sleep(2);

  execve(procname, newargv, newenviron);

  /* execve() only returns on error */
  perror("execve");
  exit(1);
}

/* queue -processes implementation*/
typedef struct ProcessList {
  pid_t procid;
  struct ProcessList *next;
  char procname[10];
  int TaskId;
} ProcessList;

struct queue {
  ProcessList *items; // array to store queue elements
  int maxsize;        // maximum capacity of the queue
  int front;          // front points to front element in the queue (if any)
  int rear;          // rear points to last element in the queue
  int size;          // current capacity of the queue
};

// Utility function to initialize queue
struct queue *newProcessQueue(int size) {
  struct queue *pt = NULL;
  pt = (struct queue *)malloc(sizeof(struct queue));

  pt->items = (ProcessList *)malloc(size * sizeof(ProcessList));
  pt->maxsize = size;
  pt->front = 0;
  pt->rear = -1;
  pt->size = 0;

  return pt;
}

// Utility function to return the size of the queue
int size(struct queue *pt) { return pt->size; }

// Utility function to check if the queue is empty or not
int isEmpty(struct queue *pt) { return !size(pt); }

// Utility function to return front element in queue
ProcessList front(struct queue *pt) {
  if (isEmpty(pt)) {
    printf("UnderFlow\nProgram Terminated\n");
    exit(EXIT_FAILURE);
  }

  return pt->items[pt->front];
}

/* function to enqueue  process */
void enqueue(struct queue *pt, ProcessList x) {
  if (size(pt) == pt->maxsize) {
    printf("OverFlow\nProgram Terminated\n");
    exit(EXIT_FAILURE);
  }

  printf("Inserting process %s  with pid %ld  and Id :%d at the end of the "
         "queue \n",
         x.procname, x.procid, x.TaskId);

  pt->rear = (pt->rear + 1) % pt->maxsize; // circular queue
  pt->items[pt->rear] = x;
  pt->size++;

  // printf("front = %d, rear = %d\n", pt->front, pt->rear);
}
```

```
/*function to dequeue process */
void dequeue(struct queue *pt) {
  if (isEmpty(pt)) // front == rear
  {
    printf("UnderFlow\nProgram Terminated\n");
    exit(EXIT_FAILURE);
  }

  printf("Removing  process %s  with pid: %ld \n", (front(pt)).procname,
         (front(pt)).procid);

  pt->front = (pt->front + 1) % pt->maxsize; // circular queue
  pt->size--;

  // printf("front = %d, rear = %d\n", pt->front, pt->rear);
}

/*function to remove item from middle of queue */
struct queue *removeFromMiddle(struct queue *pt, pid_t pid) {
  ProcessList *cu;

  if (front(pt).procid == pid) {
    cu = &(pt->items[pt->front]);

    dequeue(pt);
    printf("The removed item is the process with id :%ld and name :%s \n",
           cu->procid, cu->procname);
    return pt;
  } else {
    while (front(pt).procid != pid && !isEmpty(pt)) {
      cu = &(pt->items[pt->front]);
      dequeue(pt);
      enqueue(pt, *cu);
    }

    // remove the actual item we want
    cu = &(pt->items[pt->front]);
    dequeue(pt);

    printf("The removed item is the process with id :%ld and name :%s \n",
           cu->procid, cu->procname);
    return pt;
  }
}

int newId = 0;

/* function to print list (maybe will use !) */

void printQueue(struct queue *pt) {
  // ProcessList r=front(pt);
  bool found = false;
  int f = pt->front;
  int r = pt->rear;
  int s = sizeof(pt->items);
  int i = 0;

  printf("Number of processes :%d\n", size(pt));
  // access the items array of the queue to get the remaining processes running
  printf("--------------PROCESSES---------------------------------------------"
         "----\n");
  for (i = f; i <= r; i++) { // i< pt->size

    printf("Process with name : %s , Id: %d ,pid: %ld\n",
           (pt->items[i]).procname, (pt->items[i]).TaskId,
           (pt->items[i]).procid);
    if (i == r) {
      found = true;
      break;
    }
  }

  printf("----Runnin Process name:%s "
         "------------------------------------------\n",
         front(pt).procname);
  printf("--------------^PROCESSES^-------------------------------------------"
         "----\n");
}

/* finished Queue implementation  */

/*initializing global pointers/variables */
pid_t cur_pid = 0;
ProcessList current;
struct queue *pt = NULL;
ProcessList *process = NULL;
pid_t *pid = NULL;
pid_t shelpid;
int nproc;

pid_t pidnew;

// define boolean flag to check if process was killed from user (through the
// shell)
bool userKilled = false;

/* The Functions of the Shell */

/* Print a list of all tasks currently being scheduled.  */
static void sched_print_tasks(void) { printQueue(pt); }

/* Send SIGKILL to a task determined by the value of its
 * scheduler-specific id.
 */
```

```
static int sched_kill_task_by_id(int id) {
  pid_t tempid;
  // assert(0 && "Please fill me!");
  bool exists = false;
  printf("The process with id :%d is going to be terminated unless it is "
         "already terminated !\n",
         id);

  int f = pt->front;
  int r = pt->rear;
  int s = sizeof(pt->items);
  int i = 0;
  for (i = f; i <= r; i++) {
    if ((pt->items[i]).TaskId == id) {
      tempid = pt->items[i].procid;
      exists = true;
    }
  }

  if (exists == false) {
    printf("Whoa there !This process does not exist !");
  } else {

    kill(tempid, SIGTERM); // maybe sigterm or sigkill
    userKilled = true;
  }

  // return -ENOSYS;
}

/* Create a new task.  */
static void sched_create_task(char *executable) {
  ProcessList newprocess;

  pidnew = fork();

  if (pidnew < 0) {
    perror("error forking child ");
    exit(1);
  }

  else if (pidnew == 0) {
    doChild(executable);
  } else {

    // here define the new process and add it to the queue
    printf("Child process : %s with pid: %ld and Id:%d was created\n",
           executable, pidnew, newId);
    newprocess.procid = pidnew;
    newprocess.TaskId = newId;
    strcpy(newprocess.procname, executable);
    // pt=resize(pt);
    enqueue(pt, newprocess);

    printf("stopping...process\n");
    kill(newprocess.procid, SIGSTOP);

    newId = newId + 1;
  }
}

/* Process requests by the shell.  */
static int process_request(struct request_struct *rq) {
  switch (rq->request_no) {
  case REQ_PRINT_TASKS:
    sched_print_tasks();
    return 0;

  case REQ_KILL_TASK:
    return sched_kill_task_by_id(rq->task_arg);

  case REQ_EXEC_TASK:
    sched_create_task(rq->exec_task_arg);
    return 0;

  default:
    return -ENOSYS;
  }
}

/*
 * SIGALRM handler
 */
static void sigalrm_handler(int signum) {
  if (signum != SIGALRM) {
    fprintf(stderr, "Internal error: Called for signum %d, not SIGALRM\n",
            signum);
    exit(1);
  }
  printf("ALARM! %d seconds have passed.\n", ALARM_SEC);

  // sigstop the process and put at the end of the queue
  printf("This child [ %ld ] must now be put on hold (back of the queue).\n",
         cur_pid);
  kill(cur_pid, SIGSTOP);
}

/*
 * SIGCHLD handler
 */
static void sigchld_handler(int signum) {

  pid_t p;
```

```
int status;

if (signum != SIGCHLD) {
  fprintf(stderr, "Internal error: Called for signum %d, not SIGCHLD\n",
          signum);
  exit(1);
}

/*
 * Something has happened to one of the children.
 * We use waitpid() with the WUNTRACED flag, instead of wait(), because
 * SIGCHLD may have been received for a stopped, not dead child.
 *
 * A single SIGCHLD may be received if many processes die at the same time.
 * We use waitpid() with the WNOHANG flag in a loop, to make sure all
 * children are taken care of before leaving the handler.
 */

for (;;) {
  p = waitpid(-1, &status, WUNTRACED | WNOHANG);
  if (p < 0) {
    perror("waitpid");
    exit(1);
  }
  if (p == 0)
    break;

  explain_wait_status(p, status);

  if (WIFEXITED(status) || WIFSIGNALED(status)) {
    /* A child has died */
    printf("Parent: Received SIGCHLD, child is dead.\n");

    // remove from queue the child that finished
    // if it was the last one --> all processes finished
    if (!isEmpty(pt) && userKilled == false) {
      dequeue(pt);
    }

    else if (!isEmpty(pt) && userKilled == true) {
      pt = removeFromMiddle(pt, p);
      userKilled = false;

    }

    // this is just a guarding else -->should not go here
    else {
      printf("All processes have ended !Exiting scheduler...\n");
      exit(1);
    }

    // if after removing the last process the queue is empty then we are done
```

```
      // !
      if (isEmpty(pt)) {
        printf("All processes have ended !Exiting scheduler...\n");
        exit(1);
      }

      printf(" New child [ %ld ] is going to run now !\n", front(pt).procid);

      // wake up the next process
      cur_pid = front(pt).procid;
      kill(cur_pid, SIGCONT);
      current = front(pt);

      /* Setup the alarm again */
      if (alarm(SCHED_TQ_SEC) < 0) {
        perror("alarm");
        exit(1);
      }
    }

    if (WIFSTOPPED(status)) {
      /* A child has stopped due to SIGSTOP/SIGTSTP, etc... */
      printf("Parent: Child has been stopped. Moving right along...\n");

      // remove the child and reattach it at the end of the queue
      dequeue(pt);
      enqueue(pt, current);

      printf(" New child [ %ld ] is going to run now !\n", front(pt).procid);

      cur_pid = front(pt).procid;
      kill(cur_pid, SIGCONT);
      current = front(pt);

      /* Setup the alarm again */
      if (alarm(ALARM_SEC) < 0) {
        perror("alarm");
        exit(1);
      }
    }
  }
}

/* Disable delivery of SIGALRM and SIGCHLD. */
static void signals_disable(void) {
  sigset_t sigset;

  sigemptyset(&sigset);
  sigaddset(&sigset, SIGALRM);
  sigaddset(&sigset, SIGCHLD);
  if (sigprocmask(SIG_BLOCK, &sigset, NULL) < 0) {
    perror("signals_disable: sigprocmask");
    exit(1);
  }
}

/* Enable delivery of SIGALRM and SIGCHLD.  */
static void signals_enable(void) {
  sigset_t sigset;

  sigemptyset(&sigset);
  sigaddset(&sigset, SIGALRM);
  sigaddset(&sigset, SIGCHLD);
  if (sigprocmask(SIG_UNBLOCK, &sigset, NULL) < 0) {
    perror("signals_enable: sigprocmask");
    exit(1);
  }
}

/* Install two signal handlers.
 * One for SIGCHLD, one for SIGALRM.
 * Make sure both signals are masked when one of them is running.
 */
static void install_signal_handlers(void) {
  sigset_t sigset;
  struct sigaction sa;

  sa.sa_handler = sigchld_handler;
  sa.sa_flags = SA_RESTART;
  sigemptyset(&sigset);
  sigaddset(&sigset, SIGCHLD);
  sigaddset(&sigset, SIGALRM);
  sa.sa_mask = sigset;

  if (sigaction(SIGCHLD, &sa, NULL) < 0) {
    perror("sigaction: sigchld");
    exit(1);
  }

  sa.sa_handler = sigalrm_handler;

  if (sigaction(SIGALRM, &sa, NULL) < 0) {
    perror("sigaction: sigalrm");
    exit(1);
  }

  /*
   * Ignore SIGPIPE, so that write()s to pipes
   * with no reader do not result in us being killed,
   * and write() returns EPIPE instead.
   */
  if (signal(SIGPIPE, SIG_IGN) < 0) {
    perror("signal: sigpipe");
    exit(1);
  }
}

/* do the shell operations */
static void do_shell(char *executable, int wfd, int rfd) {
  char arg1[10], arg2[10];
  char *newargv[] = {executable, NULL, NULL, NULL};
  char *newenviron[] = {NULL};

  // write/read file descriptors
  sprintf(arg1, "%05d", wfd);
  sprintf(arg2, "%05d", rfd);

  newargv[1] = arg1;
  newargv[2] = arg2;

  raise(SIGSTOP);
  execve(executable, newargv, newenviron);

  /* execve() only returns on error */
  perror("scheduler: child: execve");
  exit(1);
}

/* creating the SHELL */
/* Create a new shell task.
 *
 * The shell gets special treatment:
 * two pipes are created for communication and passed
 * as command-line arguments to the executable.
 */
static pid_t sched_create_shell(char *executable, int *request_fd,
                                int *return_fd) {
  pid_t p;
  int pfds_rq[2], pfds_ret[2];

  if (pipe(pfds_rq) < 0 || pipe(pfds_ret) < 0) {
    perror("pipe");
    exit(1);
  }

  p = fork();
  if (p < 0) {
    perror("scheduler: fork");
    exit(1);
  }

  if (p == 0) {
    /* Child */
    close(pfds_rq[0]);
    close(pfds_ret[1]);
    do_shell(executable, pfds_rq[1], pfds_ret[0]);
    // assert(0);
  }

  /* Parent */
  close(pfds_rq[1]);
  close(pfds_ret[0]);
  *request_fd = pfds_rq[0];
  *return_fd = pfds_ret[1];
  // ADD THE SHELL
  if (p != 0) {
    return p;
  }
}

static void shell_request_loop(int request_fd, int return_fd) {
  int ret;
  struct request_struct rq;

  /*
   * Keep receiving requests from the shell.
   */
  for (;;) {
    if (read(request_fd, &rq, sizeof(rq)) != sizeof(rq)) {
      perror("scheduler: read from shell");
      fprintf(stderr, "Scheduler: giving up on shell request processing.\n");
      break;
    }

    signals_disable();
    ret = process_request(&rq);
    signals_enable();

    if (write(return_fd, &ret, sizeof(ret)) != sizeof(ret)) {
      perror("scheduler: write to shell");
      fprintf(stderr, "Scheduler: giving up on shell request processing.\n");
      break;
    }
  }
}

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

  /* Two file descriptors for communication with the shell */
  static int request_fd, return_fd;
  int i = 0;

  /* create the child processes */
  /* initialize the structures ,pointers */
  process = (ProcessList *)realloc(process, (argc) * sizeof(ProcessList));
  pid = (pid_t *)realloc(pid, (argc) * sizeof(pid_t));
  //  pt = newProcessQueue(argc); //correct
```

```
pt = newProcessQueue(100);

/* Create the shell. */
shelpid = sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd, &return_fd);
/*  add the shell to the scheduler's tasks */
process[0].procid = shelpid;
// set the id for the shell -> 0
process[0].TaskId = 0;
process[0].next = &process[1];
strcpy(process[0].procname, "shell");
enqueue(pt, process[0]);
kill(process[0].procid, SIGSTOP);

/* create all the processes ,freeze all but one */
for (i = 1; i <= argc - 1; i++) {

  pid[i] = fork();

  if (pid[i] == 0) {
    doChild(argv[i]);
  }

  if (pid[i] < 0) {

    perror("error forking child ");
    exit(1);
  }

  // add pointers to create cyclic list
  if (i != argc - 1) {
    process[i].next = &process[i + 1];
  }

  // the last points to the front
  if (i == argc - 1) {
    process[i].next = NULL;
  }

  process[i].procid = pid[i];

  // use the  "i" as task id ->used later to kill process
  process[i].TaskId = i;
  strcpy(process[i].procname, argv[i]);
  enqueue(pt, process[i]);
  kill(process[i].procid, SIGSTOP);
}

/*
 * For each of argv[1] to argv[argc - 1],
 * create a new child process, add it to the process list.
 */
```

```
  nproc = argc; /* number of proccesses goes here (argc -1
                                                      ,+1 for the shell
                 process)*/

  newId = argc; // second variable to use for creating Id for new processes

  /* Wait for all children to raise SIGSTOP before exec()ing. */
  wait_for_ready_children(nproc);

  /* Install SIGALRM and SIGCHLD handlers. */
  install_signal_handlers();

  if (nproc == 0) {
    fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
    exit(1);
  }

  /* start the first process */
  cur_pid = front(pt).procid;
  current = front(pt);

  kill(cur_pid, SIGCONT);
  if (alarm(SCHED_TQ_SEC) < 0) {
    perror("alarm");
    exit(1);
  }

  shell_request_loop(request_fd, return_fd);

  /* Now that the shell is gone, just loop forever
   * until we exit from inside a signal handler.
   */
  while (pause())
    ;

  /* Unreachable */
  fprintf(stderr, "Internal error: Reached unreachable point\n");
  return 1;
}
```

ΑΣΚ1.3

```
#include <assert.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <stdbool.h>
```

```
#include <sys/types.h>
#include <sys/wait.h>

#include "proc-common.h"
#include "request.h"

/* Compile-time parameters. */
#define SCHED_TQ_SEC 10               /* time quantum */
#define TASK_NAME_SZ 60               /* maximum size for a task's name */
#define SHELL_EXECUTABLE_NAME "shell" /* executable for shell */
#define SLEEP_SEC 1
#define ALARM_SEC 2
#define H 1
#define L 0

/* The function of the prog processses-children */
void doChild(char *procname) {
  char *newargv[] = {procname, NULL, NULL, NULL}; //& exexutable[0]
  char *newenviron[] = {NULL};

  printf("I am Process :%s, and PID = %ld\n", procname, (long)getpid());
  printf("About to replace myself with the executable %s...\n", procname);
  sleep(2);

  execve(procname, newargv, newenviron);

  /* execve() only returns on error */
  perror("execve");
  exit(1);
}

/* queue -processes implementation*/
typedef struct ProcessList {
  pid_t procid;
  struct ProcessList *next;
  char procname[10];
  int TaskId;
  int priority;
} ProcessList;

struct queue {
  ProcessList *items; // array to store queue elements
  int maxsize;        // maximum capacity of the queue
  int front;          // front points to front element in the queue (if any)
  int rear;           // rear points to last element in the queue
  int size;           // current capacity of the queue
};

// Utility function to initialize queue
struct queue *newProcessQueue(int size) {
  struct queue *pt = NULL;
  pt = (struct queue *)malloc(sizeof(struct queue));
  pt->items = (ProcessList *)malloc(size * sizeof(ProcessList));
  pt->maxsize = size;
  pt->front = 0;
  pt->rear = -1;
  pt->size = 0;

  return pt;
}

// Utility function to return the size of the queue
int size(struct queue *pt) { return pt->size; }

// Utility function to check if the queue is empty or not
int isEmpty(struct queue *pt) { return !size(pt); }

// Utility function to return front element in queue
ProcessList front(struct queue *pt) {
  if (isEmpty(pt)) {
    printf("UnderFlow\nProgram Terminated\n");
    exit(EXIT_FAILURE);
  }

  return pt->items[pt->front];
}

/* function to enqueue  process */
void enqueue(struct queue *pt, ProcessList x) {
  if (size(pt) == pt->maxsize) {
    printf("OverFlow\nProgram Terminated\n");
    exit(EXIT_FAILURE);
  }

  // printf("Inserting process %s  with pid %ld at the end of the queue \n",
  // x.procname,x.procid);

  pt->rear = (pt->rear + 1) % pt->maxsize; // circular queue
  pt->items[pt->rear] = x;
  pt->size++;

  // printf("front = %d, rear = %d\n", pt->front, pt->rear);
}

/*function to dequeue process */
void dequeue(struct queue *pt) {
  if (isEmpty(pt)) // front == rear
  {
    printf("UnderFlow\nProgram Terminated\n");
    exit(EXIT_FAILURE);
  }

  // printf("Removing  process %s  with pid: %ld \n",
  // (front(pt)).procname,(front(pt)).procid);

  pt->front = (pt->front + 1) % pt->maxsize; // circular queue
  pt->size--;

  // printf("front = %d, rear = %d\n", pt->front, pt->rear);
}

/*function to remove item from middle of queue */
struct queue *removeFromMiddle(struct queue *pt, pid_t pid) {
  ProcessList *cu;

  if (front(pt).procid == pid) {
    cu = &(pt->items[pt->front]);

    dequeue(pt);
    //  printf("The removed item is the process with id :%ld and name :%s
    //  \n",cu->procid,cu->procname);
    return pt;
  } else {
    while (front(pt).procid != pid && !isEmpty(pt)) {
      cu = &(pt->items[pt->front]);
      dequeue(pt);
      enqueue(pt, *cu);
    }

    // remove the actual item we want
    cu = &(pt->items[pt->front]);
    dequeue(pt);

    // printf("The removed item is the process with id :%ld and name :%s
    // \n",cu->procid,cu->procname);
    return pt;
  }
}

int newId = 0;

/*function to return priority */
char *priorityFinder(ProcessList p) {

  if (p.priority == H) {
    return "HIGH";
  } else if (p.priority == L) {
    return "LOW";
  }
}

/* function to print list (maybe will use !) */

void printQueue(struct queue *pt) {
  // ProcessList r=front(pt);
  bool found = false;
  int f = pt->front;
  int r = pt->rear;
  // int s=sizeof(pt->items);
  int i = 0;

  printf("Number of processes :%d\n", size(pt));
  // access the items array of the queue to get the remaining processes running
  printf("--------------PROCESSES--------------------------------------------"
         "----\n");

  if (!isEmpty(pt)) {
    for (i = f; i <= r; i++) { // i< pt->size

      printf("Process with name : %s , Id: %d ,pid: %ld ,priority %s\n",
             (pt->items[i]).procname, (pt->items[i]).TaskId,
             (pt->items[i]).procid, priorityFinder(pt->items[i]));
    }

    printf("----Runnin Process name:%s "
           "------------------------------------------\n",
           front(pt).procname);
    printf("--------------^PROCESSES^-----------------------------------------"
           "------\n");
  }
}

/* finished Queue implementation  */

/*initializing global pointers/variables */
// defining current process running for pt-lowQ queue
pid_t cur_pid = 0;
ProcessList *current;

ProcessList *process = NULL;
pid_t *pid = NULL;
pid_t shelpid;
int nproc;

pid_t pidnew;

// defining current process running for highQ queue
ProcessList *high_current;
pid_t high_pid = 0;

// define the flags for low high priority
bool Pchange = false;
bool hfound = false;

// define boolean flag to check if process was killed from user (through the
// shell)
bool userKilled = false;
```

```
// defining the new lists high and low to handle the priority
struct queue *pt = NULL;
struct queue *highQ = NULL;

/* The Functions of the Shell */

void freezeLowProcess(struct queue *pt) {
  int f = pt->front;
  int r = pt->rear;
  // int s=sizeof(pt->items);
  int i = 0;
  printf("------------------------------------------------------------------"
         "--------\n");
  printf("--------------STOPPING LOW PRIORITY PROCESSES TO HANDLE HIGH "
         "PRIORITY--------\n");
  printf("------------------------------------------------------------------"
         "--------\n");
  for (i = f; i <= r; i++) {
    kill((pt->items[i]).procid, SIGSTOP);
    printf("Stopping LOW PRIORITY process with id : %d and pid %ld\n",
           (pt->items[i]).TaskId, (pt->items[i]).procid);
  }
}

/* Print a list of all tasks currently being scheduled.  */
static void sched_print_tasks(void) {
  printQueue(pt);
  printQueue(highQ);
}

/* Send SIGKILL to a task determined by the value of its
 * scheduler-specific id.
 */
static int sched_kill_task_by_id(int id) {
  ProcessList *cur;
  pid_t tempid;
  // assert(0 && "Please fill me!");
  bool exists = false;
  printf("The process with id :%d is going to be terminated unless it is "
         "already terminated !\n",
         id);
  // check on queue 1 if the process exists
  int f = pt->front;
  int r = pt->rear;
  int i = 0;
  for (i = f; i <= r; i++) {
    if ((pt->items[i]).TaskId == id) {
      tempid = pt->items[i].procid;
      exists = true;
    }
  }
  if (exists == true) {

    // remove from pt queue and kill
    kill(tempid, SIGKILL); // or sigkill
    pt = removeFromMiddle(pt, tempid);
    // userKilled=true;

    // exit(1);
  } else {
    // check on queue2 if the process exists
    f = highQ->front;
    r = highQ->rear;
    i = 0;
    for (i = f; i <= r; i++) {
      if ((highQ->items[i]).TaskId == id) {
        tempid = highQ->items[i].procid;
        exists = true;
      }
    }
    if (exists == true) {
      // remove from highQ and kill
      kill(tempid, SIGKILL); // maybe sigterm or sigkill
      highQ = removeFromMiddle(highQ, tempid);
      // userKilled=true;
      // exit(1);
    } else {
      printf("this process does not exist !");
    }
  }
}

/* Create a new task.  */
static void sched_create_task(char *executable) {
  ProcessList newprocess;

  pidnew = fork();

  if (pidnew < 0) {
    perror("error forking child ");
    exit(1);
  }

  else if (pidnew == 0) {
    doChild(executable);
  } else {

    // here define the new process and add it to the queue
    printf("Child process : %s with pid: %ld and Id:%d was created\n",
           executable, pidnew, newId);
    newprocess.procid = pidnew;
    newprocess.TaskId = newId;
    newprocess.priority = L;
    strcpy(newprocess.procname, executable);

    // enqueue to the lowQ pt queue -->every starting has default priority low
    enqueue(pt, newprocess);

    printf("stopping...process\n");
    kill(newprocess.procid, SIGSTOP);

    newId = newId + 1;
  }
}

/* set priority to high and low */
/*function to change priority of task to ----LOW---- */
static int set_low(int id) {

  ProcessList *cur;
  pid_t tempid;
  int temp_Id;
  int prio;
  /* FROM HIGH TO LOW  ---> SEARCH ONLY IN HIGHQ */
  bool exists = false;
  printf("The process with id :%d is going to be changed to LOW priority if it "
         "is of high priority\n",
         id);

  int f = highQ->front;
  int r = highQ->rear;
  int s = sizeof(highQ->items);
  int i = 0;
  for (i = f; i <= r; i++) {
    if ((highQ->items[i]).TaskId == id) {
      cur = &(highQ->items[i]);
      tempid = highQ->items[i].procid;
      temp_Id = highQ->items[i].TaskId;
      prio = highQ->items[i].priority;
      exists = true;
    }
  }

  if (exists == false) {
    printf(
        "Whoa there !This process does not exist or has low priority already!");
  } else {
    printf("...Changing priority of process with Id:%d to LOW\n", temp_Id);

    // change priority to low and remove from highQ
    // also add to lowQ
    cur->priority = L;
    highQ = removeFromMiddle(highQ, cur->procid);
    enqueue(pt, *cur);
    // setting the change flag true
    Pchange = true;
  }
  /*
  if(Pchange==true){
     checkQueue(pt);
  }*/
}

// first change to hich should activate the ---> low stopping only high working
/*function to change priority of task to ----HIGH---- */
static int set_high(int id) {

  ProcessList *cur;
  pid_t tempid;
  int temp_Id;
  int prio;
  bool exists = false;
  printf("The process with id :%d is going to be changed to HIGH priority if "
         "it exists and has low priority\n",
         id);

  int f = pt->front;
  int r = pt->rear;
  int s = sizeof(pt->items);
  int i = 0;

  for (i = f; i <= r; i++) {
    if ((pt->items[i]).TaskId == id) {
      cur = &(pt->items[i]);
      tempid = pt->items[i].procid;
      temp_Id = pt->items[i].TaskId;
      prio = pt->items[i].priority;
      exists = true;
    }
  }

  if (exists == false) {
    printf("Whoa there !This process does not exist or is of high priority "
           "already!");

  } else {
    // change low to high -> also remove from pt(lowq) and add to highQ
    printf("...Changing priority of process with Id:%d to HIGH\n", temp_Id);
    cur->priority = H;
    pt = removeFromMiddle(pt, cur->procid);
    enqueue(highQ, *cur);

    // setting the change flag true
    // stop the low processes
    // start the first high priority --> on the first change from low to high
    freezeLowProcess(pt);
    // the only element in thw highQ
    if (size(highQ) == 1) {
      // set the pointer to the running high priority process
      high_current = cur;
      high_pid = cur->procid;
      printf("First process with pid :%ld , Id : %d , priority: HIGH in high "
             "priority starting\n ",
             high_pid, cur->TaskId, cur->priority);
      kill(high_pid, SIGCONT);

      if (alarm(SCHED_TQ_SEC) < 0) {
        perror("alarm");
        exit(1);
      }
    }
    Pchange = true;
  }
  /*
  if(Pchange==true){
      checkQueue(pt);
  }*/
}

/* Process requests by the shell.  */
static int process_request(struct request_struct *rq) {
  switch (rq->request_no) {
  case REQ_PRINT_TASKS:
    sched_print_tasks();
    return 0;

  case REQ_KILL_TASK:
    return sched_kill_task_by_id(rq->task_arg);

  case REQ_EXEC_TASK:
    sched_create_task(rq->exec_task_arg);
    return 0;

  case REQ_HIGH_TASK:
    set_high(rq->task_arg);
    return 0;

  case REQ_LOW_TASK:
    set_low(rq->task_arg);
    return 0;

  default:
    return -ENOSYS;
  }
}

/*
```

```
 * SIGALRM handler
 */
static void sigalrm_handler(int signum) {
  if (signum != SIGALRM) {
    fprintf(stderr, "Internal error: Called for signum %d, not SIGALRM\n",
            signum);
    exit(1);
  }
  printf("ALARM! %d seconds have passed.\n", ALARM_SEC);

  // run low priority only if highQ queue is empty--STOP AFTER quantum
  if (isEmpty(highQ)) {
    printf("This child [ %ld ] must now be put on hold (back of the queue).\n",
           cur_pid);
    kill(cur_pid, SIGSTOP);
  }

  // run high priority --STOP AFTER quantum
  else {
    printf("This child [ %ld ] must now be put on hold (back of the queue).\n",
           high_pid);
    kill(high_pid, SIGSTOP);
  }
}

/*
 * SIGCHLD handler
 */
static void sigchld_handler(int signum) {

  pid_t p;
  int status;

  if (signum != SIGCHLD) {
    fprintf(stderr, "Internal error: Called for signum %d, not SIGCHLD\n",
            signum);
    exit(1);
  }

  /*
   * Something has happened to one of the children.
   * We use waitpid() with the WUNTRACED flag, instead of wait(), because
   * SIGCHLD may have been received for a stopped, not dead child.
   *
   * A single SIGCHLD may be received if many processes die at the same time.
   * We use waitpid() with the WNOHANG flag in a loop, to make sure all
   * children are taken care of before leaving the handler.
   */

  for (;;) {
    p = waitpid(-1, &status, WUNTRACED | WNOHANG);
    if (p < 0) {
```

```
  perror("waitpid");
  exit(1);
}
if (p == 0)
  break;

explain_wait_status(p, status);

if (WIFEXITED(status) || WIFSIGNALED(status)) {
  // high queue has still elements to handle
  if (size(highQ) != 0) {

    /* A child has died */
    printf("Parent: Received SIGCHLD, child is dead.\n");
    dequeue(highQ);

    //  else if(userKilled==true){
    //      highQ=removeFromMiddle(highQ,p);
    //      userKilled=false;
    //  }

    // wake up the next process

    if (!isEmpty(highQ)) {
      high_pid = front(highQ).procid;
      kill(high_pid, SIGCONT);
      high_current = &highQ->items[highQ->front];

      printf(" New child [ %ld ] is going to run now !\n",
             front(highQ).procid);

      /* Setup the alarm again */
      if (alarm(SCHED_TQ_SEC) < 0) {
        perror("alarm");
        exit(1);
      }
    }
  }
  if (isEmpty(highQ)) {
    printf("All high PRIORITY processes have ended.\n");
    printf("-------------------------------------------\n");
    printf("------ACTIVATING LOW PRIORITY PROCESSES------\n");
    printf("---------------------------------------------\n");

    // here check if there are any low priority to activate -->if not exit
    // if after removing the last process the queue is empty then we are
    // done !
    if (isEmpty(pt)) {
      printf("All processes have ended !Exiting scheduler...\n");
      exit(1);

    } else { // handle the left elements in the low -pt queue
```

```
          // wake up the next process
          printf("--THE LOW PRIORITY PROCESSES ARE GOING TO CONTINUE NOW--\
n");
          cur_pid = front(pt).procid;
          kill(cur_pid, SIGCONT);
          current = &pt->items[pt->front];
        }
      }

    } // end of WIFEXITED

    if (WIFSTOPPED(status)) {

      /* A child has stopped due to SIGSTOP/SIGTSTP, etc... */
      // printf("Parent: Child has been stopped. Moving right along...\n");

      // remove the child and reattach it at the end of the queue
      if (size(highQ) != 0) {

        dequeue(highQ);
        enqueue(highQ, *high_current);

        printf(" New child [ %ld ] is going to run now !\n",
               front(highQ).procid);

        high_pid = front(highQ).procid;

        kill(high_pid, SIGCONT);
        high_current = &highQ->items[highQ->front];

        /* Setup the alarm again */
        if (alarm(ALARM_SEC) < 0) {
          perror("alarm");
          exit(1);
        }

      } else {
        // remove the child and reattach it at the end of the queue
        dequeue(pt);
        enqueue(pt, *current);

        printf(" New child [ %ld ] is going to run now !\n", front(pt).procid);

        cur_pid = front(pt).procid;
        kill(cur_pid, SIGCONT);
        current = &pt->items[pt->front];

        /* Setup the alarm again */
        if (alarm(ALARM_SEC) < 0) {
          perror("alarm");
          exit(1);
        }
      }
    } // end of WIFSTOPPED
  }
}

/* Disable delivery of SIGALRM and SIGCHLD. */
static void signals_disable(void) {
  sigset_t sigset;

  sigemptyset(&sigset);
  sigaddset(&sigset, SIGALRM);
  sigaddset(&sigset, SIGCHLD);
  if (sigprocmask(SIG_BLOCK, &sigset, NULL) < 0) {
    perror("signals_disable: sigprocmask");
    exit(1);
  }
}

/* Enable delivery of SIGALRM and SIGCHLD.  */
static void signals_enable(void) {
  sigset_t sigset;

  sigemptyset(&sigset);
  sigaddset(&sigset, SIGALRM);
  sigaddset(&sigset, SIGCHLD);
  if (sigprocmask(SIG_UNBLOCK, &sigset, NULL) < 0) {
    perror("signals_enable: sigprocmask");
    exit(1);
  }
}

/* Install two signal handlers.
 * One for SIGCHLD, one for SIGALRM.
 * Make sure both signals are masked when one of them is running.
 */
static void install_signal_handlers(void) {
  sigset_t sigset;
  struct sigaction sa;

  sa.sa_handler = sigchld_handler;
  sa.sa_flags = SA_RESTART;
  sigemptyset(&sigset);
  sigaddset(&sigset, SIGCHLD);
  sigaddset(&sigset, SIGALRM);
  sa.sa_mask = sigset;

  if (sigaction(SIGCHLD, &sa, NULL) < 0) {
    perror("sigaction: sigchld");
    exit(1);
  }

  sa.sa_handler = sigalrm_handler;
  if (sigaction(SIGALRM, &sa, NULL) < 0) {
    perror("sigaction: sigalrm");
    exit(1);
  }

  /*
   * Ignore SIGPIPE, so that write()s to pipes
   * with no reader do not result in us being killed,
   * and write() returns EPIPE instead.
   */
  if (signal(SIGPIPE, SIG_IGN) < 0) {
    perror("signal: sigpipe");
    exit(1);
  }
}

/* probably better to let them as is */
/* do the shell operations */
static void do_shell(char *executable, int wfd, int rfd) {
  char arg1[10], arg2[10];
  char *newargv[] = {executable, NULL, NULL, NULL};
  char *newenviron[] = {NULL};

  // write/read file descriptors
  sprintf(arg1, "%05d", wfd);
  sprintf(arg2, "%05d", rfd);

  newargv[1] = arg1;
  newargv[2] = arg2;

  raise(SIGSTOP);
  execve(executable, newargv, newenviron);

  /* execve() only returns on error */
  perror("scheduler: child: execve");
  exit(1);
}

/* creating the SHELL */
/* Create a new shell task.
 *
 * The shell gets special treatment:
 * two pipes are created for communication and passed
 * as command-line arguments to the executable.
 */
static pid_t sched_create_shell(char *executable, int *request_fd,
                                int *return_fd) {
  pid_t p;
  int pfds_rq[2], pfds_ret[2];

  if (pipe(pfds_rq) < 0 || pipe(pfds_ret) < 0) {
    perror("pipe");
    exit(1);
  }

  p = fork();
  if (p < 0) {
    perror("scheduler: fork");
    exit(1);
  }

  if (p == 0) {
    /* Child */
    close(pfds_rq[0]);
    close(pfds_ret[1]);
    do_shell(executable, pfds_rq[1], pfds_ret[0]);
    // assert(0);
  }

  /* Parent */
  close(pfds_rq[1]);
  close(pfds_ret[0]);
  *request_fd = pfds_rq[0];
  *return_fd = pfds_ret[1];
  // ADD THE SHELL
  if (p != 0) {
    return p;
  }
}

static void shell_request_loop(int request_fd, int return_fd) {
  int ret;
  struct request_struct rq;

  /*
   * Keep receiving requests from the shell.
   */
  for (;;) {
    if (read(request_fd, &rq, sizeof(rq)) != sizeof(rq)) {
      perror("scheduler: read from shell");
      fprintf(stderr, "Scheduler: giving up on shell request processing.\n");
      break;
    }

    signals_disable();
    ret = process_request(&rq);
    signals_enable();

    if (write(return_fd, &ret, sizeof(ret)) != sizeof(ret)) {
      perror("scheduler: write to shell");
      fprintf(stderr, "Scheduler: giving up on shell request processing.\n");
      break;
    }
  }
}

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

  /* Two file descriptors for communication with the shell */
  static int request_fd, return_fd;
  int i = 0;

  /* create the child processes */
  /* initialize the structures ,pointers */
  process = (ProcessList *)realloc(process, (argc) * sizeof(ProcessList));
  pid = (pid_t *)realloc(pid, (argc) * sizeof(pid_t));
  //  pt = newProcessQueue(argc); //correct

  // define our 3 queues
  pt = newProcessQueue(100); // low queue
  // lowQ = newProcessQueue(50);
  highQ = newProcessQueue(100); // high queue

  /* Create the shell. */
  shelpid = sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd, &return_fd);
  /*  add the shell to the scheduler's tasks */
  process[0].procid = shelpid;
  // set the id for the shell -> 0
  process[0].TaskId = 0;
  process[0].priority = L;
  process[0].next = &process[1];
  strcpy(process[0].procname, "shell");
  enqueue(pt, process[0]);
  kill(process[0].procid, SIGSTOP);

  /* create all the processes ,freeze all but one */
  for (i = 1; i <= argc - 1; i++) {

    pid[i] = fork();

    if (pid[i] == 0) {
      doChild(argv[i]);
    }

    if (pid[i] < 0) {

      perror("error forking child ");
      exit(1);
    }

    // add pointers to create cyclic list
    if (i != argc - 1) {
      process[i].next = &process[i + 1];
    }
    // the last points to the front
    if (i == argc - 1) {
      process[i].next = NULL;
    }

    process[i].procid = pid[i];
    process[i].priority = L;
    // use the  "i" as task id ->used later to kill process
    process[i].TaskId = i;
    strcpy(process[i].procname, argv[i]);
    enqueue(pt, process[i]);
    kill(process[i].procid, SIGSTOP);
  }

  /*
   * For each of argv[1] to argv[argc - 1],
   * create a new child process, add it to the process list.
   */

  nproc = argc; /* number of proccesses goes here (argc -1
                                                        ,+1 for the shell
                 process)*/

  newId = argc; // second variable to use for creating Id for new processes

  /* Wait for all children to raise SIGSTOP before exec()ing. */
  wait_for_ready_children(nproc);

  /* Install SIGALRM and SIGCHLD handlers. */
  install_signal_handlers();

  if (nproc == 0) {
    fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
    exit(1);
  }

  /* start the first process
which is the shell  */
  cur_pid = front(pt).procid;
  current = &pt->items[pt->front];

  kill(cur_pid, SIGCONT);
  if (alarm(SCHED_TQ_SEC) < 0) {
    perror("alarm");
    exit(1);
  }

  shell_request_loop(request_fd, return_fd);

  /* Now that the shell is gone, just loop forever
   * until we exit from inside a signal handler.
   */
  while (pause())
    ;

  /* Unreachable */
  fprintf(stderr, "Internal error: Reached unreachable point\n");
  return 1;
}
```