4^{η} Αναφορά στα Λειτουργικά Συστήματα

Ιωάννης Αλεξόπουλος (03117001)
Αλέξανδρος Κυριακάκης (03112163)

June 6, 2020

1 Άσκηση 1

Στο ερώτημα αυτό υλοποιήσαμε έναν round-robin scheduler με τον scheduler να αποτελεί τη γονική διεργασία και τις προς χρονοδρομολόγηση διεργασίες τα παιδιά της. Η κεντρική ιδέα στηρίζεται στη δημιουργία των κατάλληλων sighandlers οι οποίοι χειρίζονται κατάλληλα ή την εκπνοή του χρόνου (δηλαδή χειρισμός ενός sigalarm) είτε την αλλαγή κατάστασης κάποιου παιδιού (δηλαδή χειρισμός ενός sigchild). Πριν παραθέσουμε τον κώδικα και απαντήσουμε στα ερώτηματα επισημαίνουμε ότι κάνουμε χρήση του alarm(tq) για να ρυθμίσουμε ένα "ξυπνητήρι" ώστε με την εκπνοή του χρόνου tq το λειτουργικό μας να λαμβάνει ένα sigalarm. Ένα sigalarm σηματοδοτεί στην αλλαγή της εκτελούμενης διεργασίας από την επόμενη διεργασία προς εξυπηρέτηση.
Ας δούμε τον κώδικα:

1.1 Κώδικας

```
1  #include <errno.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <stdio.h>
5  #include <signal.h>
6  #include <string.h>
7  #include <assert.h>
8
9  #include <sys/wait.h>
10 #include <sys/types.h>
11
12 #include "proc-common.h"
13 #include "request.h"
14
15 /* Compile-time parameters. */
16 #define TQ 2                /* time quantum */
```

```c
#define TASK_NAME_SZ 60               /* maximum size for a
    task's name */

struct Queue_Node {
    pid_t pid;
    char *name;
    struct Queue_Node *next;

};

/* struct of queue node */
typedef struct Queue_Node node;
node *head = NULL;
node *tail = NULL;
int item_count;

/* insert a new element in the queue */
void insertq(pid_t pid, char *name) {
    node *new_node = malloc(sizeof(node));
    new_node->pid = pid;
    new_node->name = name;
    node *temp = head;
    if (temp == NULL) {
        head = new_node;
        tail = new_node;
    }
    else {
        tail->next = new_node;
        tail = new_node;
    }
    item_count++;
}


/*
 * SIGALRM handler
 */
static void sigalrm_handler(int signum)
{
    if (kill(head->pid, SIGSTOP) < 0) {
        perror("sigstop error");
        exit(1);
    }
}

/*
```

```
62   * SIGCHLD handler
63   */
64  static void sigchld_handler(int signum)
65  {
66      pid_t p;
67      int status;
68      for (;;) {
69          p = waitpid(-1, &status, WUNTRACED | WNOHANG);
70          if (p == 0)
71              break;
72          explain_wait_status(p, status);
73          if (WIFEXITED(status)) {
74              if (p != head->pid) {
75                  printf("Received exit signal not from head\n");
76                  exit(1);
77              }
78              /* Child has died */
79              printf("%d\n", item_count);
80              if (item_count > 1) {
81                  node *temp = head;
82                  head = head->next;
83                  tail->next = head;
84                  if(head->next != head) free(temp);
85                  --item_count;
86                  if (kill(head->pid, SIGCONT) < 0) {
87                      printf("pid that is ruinining it :%d\n",
                        ↪  head->pid);
88                      perror("sigcont error inside handler of
                        ↪  WIFEXITED");
89                      exit(1);
90                  }
91                  alarm(TQ);
92              }
93              else {
94                  printf("Last child has died normally\n");
95                  exit(0);
96              }
97
98          }
99          if (WIFSIGNALED(status)) {
100                 node *temp = head;
101                 int counter = 0;
102                 while(counter < item_count) {
103                     if ((temp->next)->pid == p) {
104                         --item_count;
105                         if (item_count > 0) {
```

```
106                             temp->next = (temp->next)->next;
107                             if (temp->next == tail) {
108                                 tail = temp;
109                             }
110                             if (temp->pid == head->pid) {
111                                 head = head->next;
112                                 alarm(TQ);
113                             }
114                         }
115                         else {
116                             printf("Last child has died
                                ↪  unexpectedly\n");
117                             exit(0);
118                         }
119                         // sched_print_tasks();
120                     }
121                     else {
122                         temp = temp->next;
123                         counter++;
124                     }
125                 }
126         }
127         if (WIFSTOPPED(status)) {
128             /* Child has stopped due to SIGSTOP */
129             head = head->next;
130             tail = tail->next;
131             if (kill(head->pid, SIGCONT) < 0) {
132                 perror("sigcont error inside handler of
                    ↪  WIFSTOPPED");
133                 exit(1);
134             }
135             alarm(TQ);
136         }
137     }
138 }
139
140 /* Install two signal handlers.
141  * One for SIGCHLD, one for SIGALRM.
142  * Make sure both signals are masked when one of them is running.
143  */
144 static void install_signal_handlers(void)
145 {
146  sigset_t sigset;
147  struct sigaction sa;
148
149  sa.sa_handler = sigchld_handler;
```

```
150   sa.sa_flags = SA_RESTART;
151   sigemptyset(&sigset);
152   sigaddset(&sigset, SIGCHLD);
153   sigaddset(&sigset, SIGALRM);
154   sa.sa_mask = sigset;
155   if (sigaction(SIGCHLD, &sa, NULL) < 0) {
156    perror("sigaction: sigchld");
157    exit(1);
158   }
159
160   sa.sa_handler = sigalrm_handler;
161   if (sigaction(SIGALRM, &sa, NULL) < 0) {
162    perror("sigaction: sigalrm");
163    exit(1);
164   }
165
166   /*
167    * Ignore SIGPIPE, so that write()s to pipes
168    * with no reader do not result in us being killed,
169    * and write() returns EPIPE instead.
170    */
171   if (signal(SIGPIPE, SIG_IGN) < 0) {
172    perror("signal: sigpipe");
173    exit(1);
174   }
175  }
176
177  int main(int argc, char *argv[])
178  {
179   int nproc;
180      pid_t p;
181   /*
182    * For each of argv[1] to argv[argc - 1],
183    * create a new child process, add it to the process list.
184    */
185
186   nproc = argc; /* number of proccesses goes here */
187      for (int i = 1; i < nproc; i++) {
188          p = fork();
189          if (p < 0) {
190              /* fork failed */
191              perror("fork");
192              exit(1);
193          }
194          if (p == 0) {
195              /* In child process */
```

```
196             char executable[TASK_NAME_SZ];
197             strcpy(executable, argv[i]);
198             // printf("%s\n",executable);
199             char *newargv[] = { executable, NULL, NULL, NULL };
200             char *newenviron[] = { NULL };
201             kill(getpid(), SIGSTOP);
202             execve(executable, newargv, newenviron);
203         }
204         else {
205             insertq(p, argv[i]);
206         }
207
208     }
209     //printf("did the forking business\n");
210     /* Circularize the list */
211     tail->next = head;
212  /* Wait for all children to raise SIGSTOP before exec()ing. */
213  wait_for_ready_children(nproc-1);
214
215  /* Install SIGALRM and SIGCHLD handlers. */
216  install_signal_handlers();
217     if (kill(head->pid, SIGCONT) < 0) {
218         perror("sigcont error");
219         exit(1);
220     }
221     printf("Woke up head\n");
222     alarm(TQ);
223
224  if (nproc == 0) {
225   fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
226   exit(1);
227  }
228
229
230  /* loop forever  until we exit from inside a signal handler. */
231  while (pause())
232   ;
233
234  /* Unreachable */
235  fprintf(stderr, "Internal error: Reached unreachable point\n");
236  return 1;
237 }
```

1.2 Έξοδος

```
prog
prog
My PID = 9131: Child PID = 9132 has been stopped by a signal, signo = 19
My PID = 9131: Child PID = 9133 has been stopped by a signal, signo = 19
Woke up head
prog: Starting, NMSG = 200, delay = 56
prog[9132]: This is message 0
prog[9132]: This is message 1
prog[9132]: This is message 2
prog[9132]: This is message 3
prog[9132]: This is message 4
prog[9132]: This is message 5
prog[9132]: This is message 6
prog[9132]: This is message 7
prog[9132]: This is message 8
prog[9132]: This is message 9
prog[9132]: This is message 10
prog[9132]: This is message 11
prog[9132]: This is message 12
prog[9132]: This is message 13
prog[9132]: This is message 14
prog[9132]: This is message 15
prog[9132]: This is message 16
prog[9132]: This is message 17
prog[9132]: This is message 18
prog[9132]: This is message 19
prog[9132]: This is message 20
prog[9132]: This is message 21
prog[9132]: This is message 22
My PID = 9131: Child PID = 9132 has been stopped by a signal, signo = 19
prog: Starting, NMSG = 200, delay = 37
prog[9133]: This is message 0
prog[9133]: This is message 1
prog[9133]: This is message 2
prog[9133]: This is message 3
prog[9133]: This is message 4
prog[9133]: This is message 5
prog[9133]: This is message 6
prog[9133]: This is message 7
prog[9133]: This is message 8
prog[9133]: This is message 9
```

1.3 Ερώτημα 1

Κατόπιν αρκετής ενασχόλησης καταλάβαμε ότι στην ουσία ο τρόπος διαχείρισης των σημάτων στηρίζεται στην χρήση μιας μάσκας συγκεκριμένου πλήθους bit. Θεωρούμε ότι κάθε bit είναι στην ουσία μία ένδειξη για το αν έχει έρθει ένα συγκεκριμένο σήμα προς εξυπηρέτηση. Έχουμε δηλαδή ένα-προς-ένα αντιστοιχία των bit της μάσκας με το είδος της εξυπηρέτησης που πρέπει να γίνει. Άρα αν έχουμε αλλεπάλληλα σήματα η πληροφορία "buff-άρεται" στην μάσκα, ενημερώνει δηλαδή το αντίστοιχο bit και αφού τελειώσει η εξυπηρέτηση ο έλεγχος πάει προς την διαχείριση της επόμενης αίτησης. Συνεπώς με τη χρήση της install_signal_handlers() γίνονται mask τα σήματα SIGCHLD και SIGALRM οπότε με την εκκίνηση του ενός handler που εξυπηρετεί ένα σήμα, απενεργοποιείται/"παγώνει" η μάσκα/εξυπηρέτηση όλων των σημάτων. Επομένως, αν έρθει ένα νέο σήμα αυτό περιμένει έως ότου τελειώσει η εξυπηρέτηση του τρέχοντος για να εξυπηρετηθεί το ίδιο

1.4 Ερώτημα 2

Το SIGCHLD είναι ένα σήμα που λαμβάνει μια γονική διεργασία όταν αλλάξει κατάσταση μια διεργασία-παιδί της. Στην προκειμένη περίπτωση η γονική διεργασία είναι ο scheduler μας και παιδιά της είναι οι διεργασίες προς εξυπηρέτηση. Στον κώδικά μας η διεργασία αυτή μπορεί να είναι οιαδήποτε θεωρητικά αφού εκτελούμε την waitpid με όρισμα -1. Αν θεωρήσουμε ότι εξωτερικά του scheduler δε μπορούν να σταλούν σήματα με χρήση μιας κατάλληλης εντολής kill τότε η αλλαγή κατάστασης ενός παιδιού γίνεται ή με την εκπνοή του χρόνου του alarm (WIFSTOPPED) είτε με τη διεκπεραίωση/περάτωση του σκοπού της διεργασίας παιδί (δηλαδή η διεργασία να τερμάτισε κανονικά με WIFEXITED). Αν πάλι θεωρήσουμε ότι δύναται να σταλεί ένα εξωτερικό σήμα (π.χ. SIGKILL) σε ένα παιδί τότε ο έλεγχος της γονικής διεργασίας μετατίθεται στο sigchld_handler με μηδενική τιμή στο macro WIFSIGNALED. Εκεί έχουμε την εξής διάκριση: Πρώτον, σε περίπτωση λήψης σήματος SIGCHLD με WIFSTOPPED flag ελέγχουμε αν η λίστα έχει μόνο την κεφαλή ως στοιχείο και άρα πρέπει αυτόματα να λάβει σήμα συνέχισης ή αν η λίστα έχει κι άλλα στοιχεία οπότε δίνουμε SIGCONT στην επόμενη διεργασία.
Δεύτερον, αν λήφθηκε SIGCHLD λόγω αποστολής σήματος SIGKILL τότε ο έλεγχος διακρίνει διάφορες περιπτώσεις. Αν η λίστα έχει μοναδική διεργασία τότε έχουμε exit και κατάλληλο μήνυμα που περιγράφει το error, αλλιώς αν στάλθηκε SIGKILL στην κεφαλή μιας λίστας με αρκετά στοιχεία τότε μεταβιβάζουμε τον έλεγχο στο επόμενο στοιχείο δίνοντας κατάλληλο σήμα SIGCONT. Τέλος, διακρίνουμε αν το στοιχείο στο οποίο απευθύνεται το σήμα τερματισμού είναι στην ουρά ή κάπου στους ενδιάμεσους κόμβους. Και στις δύο αυτές περιπτώσεις ενημερώνουμε κατάλληλα. Αν, ωστόσο, το σήμα απευθύνεται σε μη υπαρκτή διεργασία δεν έχουμε κανένα πρόβλημα γιατί δε θα προξενηθεί κάποια αλλαγή στην κατάσταση των παιδιών του χρονοδρομολογητή.
Τέλος, αν λάβουμε σήμα SIGCHLD με WIFEXITED γνωρίζουμε από την σχεδίαση του χρονοδρομολογητή ότι κάτι τέτοιο μπορεί να συμβεί μόνο από την "τρέχουσα"

διεργασία που βρίσκεται στην κεφαλή της λίστας (Σε αντίθετη περίπτωση έχουμε σφάλμα) και αναλόγως αν απομένουν επιπλέον εργασίες μεταβιβάζουμε τον έλεγχο στο επόμενο στοιχείο της λίστας ή τερματίζουμε τον χρονοδρομολογητή.

1.5 Ερώτημα 3

Το πρόβλημα έγκειται στο ότι με τη χρήση του SIGALRM και μόνο για τη χρονοδρομολόγηση, δεν εξασφαλίζεται η παύση μιας διεργασίας-παιδιού αν δε ληφθεί το σήμα SIGCHLD. Με το συνδυασμό, λοιπόν, των δύο σημάτων εξασφαλίζεται η για μια νέα ενέργεια (π.χ. αλλαγή της διεργασίας που εξυπηρετείται με μία άλλη) η τρέχουσα διεργασία θα έχει σταματήσει. Επομένως, δε θα βρεθούμε σε δυσερμήνευτες και μη δίκαιες από άποψη χρόνου εξυπηρέτησης συμπεριφορές.

2 Άσκηση 2

Ας δούμε τον κώδικα:

2.1 Κώδικας

```
1  #include <errno.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <stdio.h>
5  #include <signal.h>
6  #include <string.h>
7  #include <assert.h>
8  #include <sys/wait.h>
9  #include <sys/types.h>
10 #include "proc-common.h"
11 #include "request.h"
12
13 /* Compile-time parameters. */
14 #define TQ 2                    /* time quantum */
15 #define TASK_NAME_SZ 60            /* maximum size for a
   ↪  task's name */
16 #define SHELL_EXECUTABLE_NAME "shell" /* executable for shell */
17
18
19 /* struct of queue node */
20 struct Queue_Node {
21  int pnum;
22     pid_t pid;
23     char *name;
24     struct Queue_Node *next;
```

```
25
26  };
27  typedef struct Queue_Node node;
28
29  node *head = NULL;    /* head of queue */
30  node *tail = NULL;    /* tail of queue */
31
32  int item_count;     /* item count */
33  int total;       /* all processes created */
34
35
36  void *safe_malloc(size_t size)
37  {
38   void *p;
39
40   if ((p = malloc(size)) == NULL) {
41    fprintf(stderr, "Out of memory, failed to allocate %zd
     ↪  bytes\n",
42     size);
43    exit(1);
44   }
45
46   return p;
47  }
48
49  /* insert a new element in the queue */
50  void insertq(pid_t pid, char *name, int pnum)
51  {
52      node *new_node = safe_malloc(sizeof(node));
53      new_node->pid = pid;
54      new_node->name = name;
55      new_node->pnum = pnum;
56      new_node->next = NULL;
57      node *temp = head;
58      if (temp == NULL) {
59          head = new_node;
60          tail = new_node;
61      }
62      else {
63          tail->next = new_node;
64          tail = new_node;
65      }
66      item_count++;
67  }
68
69  /* Print a list of all tasks currently being scheduled.  */
```

```
70  static void sched_print_tasks(void)
71  {
72   node *temp;
73   printf("--- PRINTING PROCESS LIST: ---\n");
74   temp = head;
75   int count = 0;
76   while (count < item_count) {
77    if (temp == head) {
78     printf("CURRENT RUNNING PROCESS:");
79    }
80    if (temp !=head) {
81     printf("                         ");
82    }
83       printf("***** PNUM=%d, PID=%d ,P_EX_NAME=%s *****\n",
84       temp->pnum , temp->pid , temp->name);
85       temp = temp->next;
86       count++;
87   }
88  }
89
90  /* Send SIGTERM to a task determined by the value of its
91   * scheduler-specific id.
92   */
93  static int sched_kill_task_by_id(int id)
94  {
95   //id given is NOT the PID but PNUM!
96   node *temp;
97   //first we have to make sure that the proc-list will be
     ↪  informed.
98   //thus we check each element to find the corresponding pnumde
     ↪  *temp;
99   //printf("Send SIGTERM to procedure with PNUM:%d\n",id);
100  temp = head;
101  int counter = 0;
102  while(counter < item_count) {
103   if ((temp->next)->pnum == id ) {
104    --item_count;
105    if (kill((temp->next)->pid, SIGTERM) < 0) {
106           perror("SIGTERM error");
107           exit(1);
108          }
109          if (item_count > 0) {
110           temp->next = (temp->next)->next;
111           if (temp->next == tail) {
112            tail = temp;
113           }
```

```
114             if (id == 0) {
115              head = head->next;
116              alarm(TQ);
117             }
118            }
119            else {
120             printf("All killed by shell\n");
121             exit(0);
122            }
123            // sched_print_tasks();
124            return 0;
125       }
126       else {
127        temp = temp->next;
128        counter++;
129       }
130   }
131   printf("There is no such task to TERMINATE!\n");
132   return 0;
133  }
134
135
136  /* Create a new task.  */
137  static void sched_create_task(char *executable)
138  {
139          pid_t p;
140          node *curr;
141          node *temp;
142          char *newargv[] = {executable,NULL,NULL,NULL};
143          char *newenviron[] = {NULL};
144
145          printf("about to create a new process/task\n");
146          curr = safe_malloc(sizeof(node));
147          temp = head;
148          sched_print_tasks();
149          int i = 0;
150          printf("ITEM_COUNT = %d\n", item_count);
151          /* add new element to the end of the queue */
152          while (i < item_count - 1) {
153           printf("temp->pnum: %d\n", temp->pnum);
154           temp = temp->next;
155           i++;
156          }
157          temp->next = curr;
158          curr->pnum = total++;
159          item_count++;
```

```
160             curr->name = strdup(executable);
161             printf("%s\n", curr->name);
162             curr->next = head;
163             tail = curr;
164             p = fork();
165             if (p < 0) {
166                 perror("fork");
167                 exit(1);
168             }
169             else if (p == 0) {
170              change_pname(executable);
171                 printf("I am %d,PID = %d\n",curr->pnum, getpid());
172                 printf("About to be replaced with the executable
                    ↪  %s..\n", executable);
173                 raise(SIGSTOP);
174                 execve(executable, newargv, newenviron);
175                 perror("execve");
176                 exit(1);
177             }
178             else {
179              curr->pid = p;
180              sched_print_tasks();
181             }
182 
183 }
184 
185 /* Process requests by the shell.  */
186 static int process_request(struct request_struct *rq)
187 {
188  switch (rq->request_no) {
189   case REQ_PRINT_TASKS:
190    sched_print_tasks();
191    return 0;
192 
193   case REQ_KILL_TASK:
194    return sched_kill_task_by_id(rq->task_arg);
195 
196   case REQ_EXEC_TASK:
197    sched_create_task(rq->exec_task_arg);
198    return 0;
199 
200   default:
201    return -ENOSYS;
202  }
203 }
204 
```

```
205 /*
206  * SIGALRM handler
207  */
208 static void sigalrm_handler(int signum)
209 {
210     if (kill(head->pid, SIGSTOP) < 0) {
211      printf("%d\n", head->pid);
212         perror("SIGSTOP error");
213         exit(1);
214     }
215 
216 }
217 
218 /*
219  * SIGCHLD handler
220  */
221 static void sigchld_handler(int signum)
222 {
223     pid_t p;
224     int status;
225     for (;;) {
226         p = waitpid(-1, &status, WUNTRACED | WNOHANG);
227         if (p == 0)
228             break;
229         explain_wait_status(p, status);
230         /* Ignore SIGCHLD of process that died from SIGTERM */
231         if (WIFEXITED(status)) {
232             /* Child has died */
233             printf("INSIDE WIFEXITED ITEM COUNT: %d\n",
                ↪  item_count);
234             if (item_count > 1) {
235              --item_count;
236                 node *temp = head;
237                 head = head->next;
238                 tail->next = head;
239                 if(head->next != head)
240                  free(temp);
241                 sched_print_tasks();
242                 if (kill(head->pid, SIGCONT) < 0) {
243                     printf("pid that is ruinining it :%d\n",
                        ↪  head->pid);
244                     perror("sigcont error inside handler of
                        ↪  WIFEXITED");
245                     exit(1);
246                 }
247                 alarm(TQ);
```

```
248             }
249             // else if (item_count == 1) {
250             //  if (kill(head->pid, SIGCONT) < 0) {
251             //          printf("pid that is ruinining it :%d\n",
                ↪  head->pid);
252             //          perror("sigcont error inside handler of
                ↪  WIFEXITED");
253             //          exit(1);
254             //      }
255             //      alarm(TQ);
256             // }
257             else {
258                 printf("Last child has died normally\n");
259                 exit(0);
260             }
261       }
262           if (WIFSTOPPED(status)) {
263               /* Child has stopped due to SIGSTOP */
264               printf("INSIDE WIFSTOPPED ITEM COUNT: %d\n",
                  ↪  item_count);
265               if (head->pid == p) {
266                printf("INSIDE IF ----------------\n");
267                if (item_count > 1) {
268                 head = head->next;
269                 tail = tail->next;
270                }
271                if (kill(head->pid, SIGCONT) < 0) {
272                    perror("sigcont error inside handler of
                       ↪  WIFSTOPPED");
273                    exit(1);
274                }
275            }
276               alarm(TQ);
277           }
278       }
279   }
280
281   /* Disable delivery of SIGALRM and SIGCHLD. */
282   static void signals_disable(void)
283   {
284    sigset_t sigset;
285
286    sigemptyset(&sigset);
287    sigaddset(&sigset, SIGALRM);
288    sigaddset(&sigset, SIGCHLD);
289    if (sigprocmask(SIG_BLOCK, &sigset, NULL) < 0) {
```

```
290     perror("signals_disable: sigprocmask");
291     exit(1);
292    }
293   }
294
295   /* Enable delivery of SIGALRM and SIGCHLD.  */
296   static void signals_enable(void)
297   {
298    sigset_t sigset;
299
300    sigemptyset(&sigset);
301    sigaddset(&sigset, SIGALRM);
302    sigaddset(&sigset, SIGCHLD);
303    if (sigprocmask(SIG_UNBLOCK, &sigset, NULL) < 0) {
304     perror("signals_enable: sigprocmask");
305     exit(1);
306    }
307   }
308
309
310   /* Install two signal handlers.
311    * One for SIGCHLD, one for SIGALRM.
312    * Make sure both signals are masked when one of them is running.
313    */
314   static void install_signal_handlers(void)
315   {
316    sigset_t sigset;
317    struct sigaction sa;
318    sa.sa_handler = sigchld_handler;
319    sa.sa_flags = SA_RESTART;
320    sigemptyset(&sigset);
321    sigaddset(&sigset, SIGCHLD);
322    sigaddset(&sigset, SIGALRM);
323    sa.sa_mask = sigset;
324    if (sigaction(SIGCHLD, &sa, NULL) < 0) {
325     perror("sigaction: sigchld");
326     exit(1);
327    }
328    sa.sa_handler = sigalrm_handler;
329    if (sigaction(SIGALRM, &sa, NULL) < 0) {
330     perror("sigaction: sigalrm");
331     exit(1);
332    }
333    /*
334     * Ignore SIGPIPE, so that write()s to pipes
335     * with no reader do not result in us being killed,
```

```
336     * and write() returns EPIPE instead.
337     */
338    if (signal(SIGPIPE, SIG_IGN) < 0) {
339     perror("signal: sigpipe");
340     exit(1);
341    }
342   }
343
344   static void do_shell(char *executable, int wfd, int rfd)
345   {
346    char arg1[10], arg2[10];
347    char *newargv[] = { executable, NULL, NULL, NULL };
348    char *newenviron[] = { NULL };
349    sprintf(arg1, "%05d", wfd);
350    sprintf(arg2, "%05d", rfd);
351    newargv[1] = arg1;
352    newargv[2] = arg2;
353    raise(SIGSTOP);
354    execve(executable, newargv, newenviron);
355    /* execve() only returns on error */
356    perror("scheduler: child: execve");
357    exit(1);
358   }
359
360   /* Create a new shell task.
361    *
362    * The shell gets special treatment:
363    * two pipes are created for communication and passed
364    * as command-line arguments to the executable.
365    */
366   static void sched_create_shell(char *executable, int *request_fd,
       ↪  int *return_fd)
367   {
368    pid_t p;
369    int pfds_rq[2], pfds_ret[2];
370    if (pipe(pfds_rq) < 0 || pipe(pfds_ret) < 0) {
371     perror("pipe");
372     exit(1);
373    }
374    p = fork();
375    if (p < 0) {
376     perror("scheduler: fork");
377     exit(1);
378    }
379    if (p == 0) {
380     /* Child */
```

```
381     close(pfds_rq[0]);
382     close(pfds_ret[1]);
383     change_pname(executable);
384     do_shell(executable, pfds_rq[1], pfds_ret[0]);
385     assert(0);
386    }
387    /* Parent */
388    if(head->pnum == 0)
389     head->pid = p;
390    close(pfds_rq[1]);
391    close(pfds_ret[0]);
392    *request_fd = pfds_rq[0];
393    *return_fd = pfds_ret[1];
394   }
395
396   static void shell_request_loop(int request_fd, int return_fd)
397   {
398    int ret;
399    struct request_struct rq;
400
401    /*
402     * Keep receiving requests from the shell.
403     */
404    for (;;) {
405     if (read(request_fd, &rq, sizeof(rq)) != sizeof(rq)) {
406      perror("scheduler: read from shell");
407      fprintf(stderr, "Scheduler: giving up on shell request
         ↪  processing.\n");
408      break;
409     }
410     printf("read from pipe and disabling signals\n");
411     signals_disable();
412     ret = process_request(&rq);
413     signals_enable();
414
415     if (write(return_fd, &ret, sizeof(ret)) != sizeof(ret)) {
416      perror("scheduler: write to shell");
417      fprintf(stderr, "Scheduler: giving up on shell request
         ↪  processing.\n");
418      break;
419     }
420    }
421   }
422
423   int main(int argc, char *argv[])
424   {
```

```
425   int nproc;
426   pid_t p;
427   /* Two file descriptors for communication with the shell */
428   static int request_fd, return_fd;
429
430   head = safe_malloc(sizeof(node));
431   tail = head;
432
433   head->pnum = 0;
434   head->name = SHELL_EXECUTABLE_NAME;
435   head->next = NULL;
436   item_count++;
437   /* Create the shell. */
438   sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd,
      ↪  &return_fd);
439   /*
440    * For each of argv[1] to argv[argc - 1],
441    * create a new child process, add it to the process list.
442    */
443   nproc = argc; /* number of proccesses goes here */
444      for (int i = 1; i < nproc; i++) {
445          p = fork();
446          if (p < 0) {
447              /* fork failed */
448              perror("fork");
449              exit(1);
450          }
451          if (p == 0) {
452              /* In child process */
453              char executable[TASK_NAME_SZ];
454              strcpy(executable, argv[i]);
455  //          printf("%s\n", executable);
456              change_pname(executable);
457              char *newargv[] = { executable, NULL, NULL, NULL };
458              char *newenviron[] = { NULL };
459              kill(getpid(), SIGSTOP);
460              execve(executable, newargv, newenviron);
461          }
462          else {
463              insertq(p, argv[i], i);
464          }
465      }
466      total = item_count;
467
468      /* Circularize the list */
469      tail->next = head;
```

```
470
471   /* Wait for all children to raise SIGSTOP before exec()ing. */
472   wait_for_ready_children(nproc);
473
474   // show_pstree(getpid());
475
476   /* Install SIGALRM and SIGCHLD handlers. */
477   install_signal_handlers();
478
479   if (nproc == 0) {
480    fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
481    exit(1);
482   }
483
484
485   if (kill(head->pid, SIGCONT) < 0) {
486          perror("sigcont error");
487          exit(1);
488      }
489
490      alarm(TQ);
491
492
493   shell_request_loop(request_fd, return_fd);
494   if (kill(head->pid, SIGCONT) < 0) {
495       perror("sigcont error");
496       exit(1);
497      }
498   alarm(TQ);
499   /* Now that the shell is gone, just loop forever
500    * until we exit from inside a signal handler.
501    */
502   while (pause())
503    ;
504
505   /* Unreachable */
506   fprintf(stderr, "Internal error: Reached unreachable point\n");
507   return 1;
508  }
```

2.2 Έξοδος

```
My PID = 15622: Child PID = 15623 has been stopped by a signal, signo = 19
My PID = 15622: Child PID = 15624 has been stopped by a signal, signo = 19
My PID = 15622: Child PID = 15625 has been stopped by a signal, signo = 19
```

```
This is the Shell. Welcome.

Shell> p
Shell: issuing request...
Shell: receiving request return value...
read from pipe and disabling signals
--- PRINTING PROCESS LIST: ---
CURRENT RUNNING PROCESS:***** PNUM=0, PID=15623 ,P_EX_NAME=shell *****
                        ***** PNUM=1, PID=15624 ,P_EX_NAME=prog *****
                        ***** PNUM=2, PID=15625 ,P_EX_NAME=prog *****
Shell> My PID = 15622: Child PID = 15623 has been stopped by a signal, signo = 19
INSIDE WIFSTOPPED ITEM COUNT: 3
INSIDE IF ----------------
prog: Starting, NMSG = 200, delay = 135
prog[15624]: This is message 0
prog[15624]: This is message 1
prog[15624]: This is message 2
prog[15624]: This is message 3
prog[15624]: This is message 4
prog[15624]: This is message 5
prog[15624]: This is message 6
prog[15624]: This is message 7
prog[15624]: This is message 8
prog[15624]: This is message 9
My PID = 15622: Child PID = 15624 has been stopped by a signal, signo = 19
INSIDE WIFSTOPPED ITEM COUNT: 3
INSIDE IF ----------------
prog: Starting, NMSG = 200, delay = 116
prog[15625]: This is message 0
prog[15625]: This is message 1
prog[15625]: This is message 2
prog[15625]: This is message 3
prog[15625]: This is message 4
prog[15625]: This is message 5
prog[15625]: This is message 6
prog[15625]: This is message 7
prog[15625]: This is message 8
prog[15625]: This is message 9
prog[15625]: This is message 10
prog[15625]: This is message 11
My PID = 15622: Child PID = 15625 has been stopped by a signal, signo = 19
```

2.3 Ερώτημα 1

Όπως φαίνεται και στην παραπάνω έξοδο δίνοντας την εντολή 'p' στον φλοιό, εμφανίζεται η λίστα με τις προς εξυπηρέτηση διεργασίες, όπου το head της δεν

είναι άλλο από τον φλοιό, δηλαδή της διεργασίας που εξυπηρετείται τη δεδομένη στιγμή που λαμβάνουμε την έξοδο. Σίγουρα μπορεί να δοθεί η εντολή p και σε κάποια φάση που ο φλοιός δεν είναι υπό εκτέλεση, να κρατηθεί και όταν έρθει η σειρά του φλοιού έχουμε την εκτέλεση της εντολής που κρατήθηκε προηγουμένως. Γενικά το να μην τυπωθεί πρώτος ο shell αλλά μια άλλη διεργασία, σημαίνει ότι πάει να γίνει εκτύπωση της λίστας σε χρονικό σημείο όπου οριακά έχει δοθεί η σειρά στο επόμενο στοιχείο της(έχει αλλάξει δηλαδή το head πριν εκτελεστεί η συνάρτηση signals_disable()). Βεβαίως κάτι τέτοιο είναι δύσκολο να το πετύχουμε λόγω του χειρισμού των σημάτων που έχουμε στον παραπάνω κώδικα (και των κωδίκων που αυτό ενέχει).

2.4 Ερώτημα 2

Ο λόγος που κάνουμε disable(), υλοποίηση ενός αιτήματος και έπειτα enable() γίνεται προκειμένου όσα σήματα έρθουν κατά τη διάρκεια της υλοποίησης ενός αιτήματος, να μην επηρεάσουν την εξυπηρέτηση του request. Αφού τελικά εξυπηρετηθεί, τότε τα σήματα που λήφθηκαν όσο είχαμε disable ενημέρωσαν τη μάσκα που περιγράφαμε προηγουμένως, όσο είχαμε την εξυπηρέτηση του request. Με την επανεπίτρεψη των σημάτων έχουμε και την ανάλογη εξυπηρέτησή τους.

3 Άσκηση 3

Ας δούμε τον κώδικα:

3.1 Κώδικας

```
1  #include <errno.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <stdio.h>
5  #include <signal.h>
6  #include <string.h>
7  #include <assert.h>
8  #include <stdbool.h>
9  #include <sys/wait.h>
10 #include <sys/types.h>
11 #include "proc-common.h"
12 #include "request.h"
13 /*
14 * Compile-time parameters.
15  */
16 #define TQ 2                    /* time quantum */
17 #define TASK_NAME_SZ 60             /* maximum size for a
   ↪  task's name */
```

```c
#define SHELL_EXECUTABLE_NAME "shell" /* executable for shell */


/* struct of queue node */
struct Queue_Node {
 int pnum;
    pid_t pid;
    char *name;
    bool high;
    struct Queue_Node *next;

};
typedef struct Queue_Node node;

node *head = NULL;    /* head of queue */
node *tail = NULL;    /* tail of queue */

int item_count;     /* item count */
int total;       /* all processes created */
int highcnt;
int lowcnt;

void *safe_malloc(size_t size)
{
 void *p;

 if ((p = malloc(size)) == NULL) {
  fprintf(stderr, "Out of memory, failed to allocate %zd
   ↪  bytes\n",
   size);
  exit(1);
 }

 return p;
}

/* insert a new element in the queue */
void insertq(pid_t pid, char *name, int pnum)
{
    node *new_node = safe_malloc(sizeof(node));
    new_node->pid = pid;
    new_node->name = name;
    new_node->pnum = pnum;
    new_node->next = NULL;
    new_node->high = false;
    node *temp = head;
```

```
63      if (temp == NULL) {
64          head = new_node;
65          tail = new_node;
66      }
67      else {
68          tail->next = new_node;
69          tail = new_node;
70      }
71      item_count++;
72  }
73
74  /* Print a list of all tasks currently being scheduled.  */
75  static void sched_print_tasks(void)
76  {
77   node *temp;
78   printf("--- PRINTING PROCESS LIST: ---\n");
79   temp = head;
80   int count = 0;
81   while (count < item_count) {
82    if (temp == head) {
83     printf("CURRENT RUNNING PROCESS:");
84    }
85    if (temp !=head) {
86     printf("                         ");
87    }
88       printf("***** PNUM=%d, PID=%d ,P_EX_NAME=%s, PRIORITY:%s
       ↪  *****\n",
89       temp->pnum , temp->pid , temp->name, temp->high ? "HIGH" :
       ↪  "LOW");
90       temp = temp->next;
91       count++;
92   }
93  }
94
95  /* Send SIGTERM to a task determined by the value of its
96   * scheduler-specific id.
97   */
98  static int sched_kill_task_by_id(int id)
99  {
100  //id given is NOT the PID but PNUM!
101  node *temp;
102  //first we have to make sure that the proc-list will be
     ↪  informed.
103  //thus we check each element to find the corresponding pnumde
     ↪  *temp;
104  //printf("Send SIGTERM to procedure with PNUM:%d\n",id);
```

```
105   temp = head;
106   int counter = 0;
107   while(counter < item_count) {
108    if ((temp->next)->pnum == id ) {
109     --item_count;
110     if (temp->next->high) {
111      highcnt--;
112     }
113     else {
114      lowcnt--;
115     }
116     if (kill((temp->next)->pid, SIGTERM) < 0) {
117            perror("SIGTERM error");
118            exit(1);
119           }
120           if (item_count > 0) {
121            temp->next = (temp->next)->next;
122            if (temp->next == tail) {
123             tail = temp;
124            }
125            if (id == 0) {
126             head = head->next;
127            }
128            //head = (id == 0) ? head->next : head;
129       // printf("head:pnum %d\n", head->pnum);
130       // printf("head->next->pnum %d\n", (head->next)->pnum);
131       // printf("head->next->next->pnum %d\n",
          ↪  (head->next)->next->pnum);
132           }
133           else {
134            printf("All killed by shell\n");
135            exit(0);
136           }
137           // sched_print_tasks();
138           return 0;
139       }
140       else {
141        temp = temp->next;
142        counter++;
143       }
144   }
145   printf("There is no such task to TERMINATE!\n");
146   return 0;
147  }
148
149
```

```
150 /* Create a new task.  */
151 static void sched_create_task(char *executable)
152 {
153         pid_t p;
154         node *curr;
155         node *temp;
156         char *newargv[] = {executable,NULL,NULL,NULL};
157         char *newenviron[] = {NULL};
158 
159         printf("about to create a new process/task\n");
160         curr = safe_malloc(sizeof(node));
161         temp = head;
162         // sched_print_tasks();
163         int i = 0;
164         printf("ITEM_COUNT = %d\n", item_count);
165         /* add new element to the end of the queue */
166         while (i < item_count - 1) {
167          printf("temp->pnum: %d\n", temp->pnum);
168          temp = temp->next;
169          i++;
170         }
171         temp->next = curr;
172         curr->pnum = total++;
173         item_count++;
174         lowcnt++;
175         curr->name = strdup(executable);
176         printf("%s\n", curr->name);
177         curr->next = head;
178         curr->high = false;
179         tail = curr;
180         p = fork();
181         if (p < 0) {
182             perror("fork");
183             exit(1);
184         }
185         else if (p == 0) {
186             printf("I am %d,PID = %d\n",curr->pnum, getpid());
187             printf("About to be replaced with the executable
                ↪  %s..\n", executable);
188             raise(SIGSTOP);
189             execve(executable, newargv, newenviron);
190             perror("execve");
191             exit(1);
192         }
193         else {
194          curr->pid = p;
```

```
195             sched_print_tasks();
196             }
197
198     }
199     static void sched_high_prior_task(int id)
200     {
201             node *temp;
202             temp = head;
203             int i = 0;
204             printf("WE ENTERED HIGH PRIORITIZE, head
                ↪  pid=%d\n",temp->pid);
205             while(i < item_count){
206                     if (temp->pnum == id) {
207                      if (temp->high)
208                       break;
209                      temp->high = true;
210                      highcnt++;
211                      lowcnt--;
212                      break;
213                     }
214                     temp = temp->next;
215                     i++;
216              }
217             printf("THE LIST AFTER HIGH PRIORITIZE:\n");
218             sched_print_tasks();
219     }
220
221     static void sched_low_prior_task(int id)
222     {
223             node *temp;
224             temp = head;
225             int i = 0;
226             printf("WE ENTERED LOW PRIORITIZE, head
                ↪  pid=%d\n",temp->pid);
227             while (i < item_count) {
228                     if (temp->pnum == id) {
229                      if (!temp->high)
230                       break;
231                    temp->high = false;
232                    lowcnt++;
233                    highcnt--;
234                       break;
235                     }
236                     temp = temp->next;
237                     i++;
238              }
```

```
239         printf("THE LIST AFTER LOW PRIORITIZE:\n");
240         sched_print_tasks();
241 }
242
243 /* Process requests by the shell.  */
244 static int process_request(struct request_struct *rq)
245 {
246  switch (rq->request_no) {
247   case REQ_PRINT_TASKS:
248    sched_print_tasks();
249    return 0;
250
251   case REQ_KILL_TASK:
252    return sched_kill_task_by_id(rq->task_arg);
253
254   case REQ_EXEC_TASK:
255    sched_create_task(rq->exec_task_arg);
256    return 0;
257
258   case REQ_HIGH_TASK:
259    sched_high_prior_task(rq->task_arg);
260    return 0;
261
262   case REQ_LOW_TASK:
263    sched_low_prior_task(rq->task_arg);
264    return 0;
265
266   default:
267    return -ENOSYS;
268  }
269 }
270
271 /*
272  * SIGALRM handler
273  */
274 static void sigalrm_handler(int signum)
275 {
276     if (kill(head->pid, SIGSTOP) < 0) {
277      printf("%d\n", head->pid);
278         perror("SIGSTOP error");
279         exit(1);
280     }
281
282 }
283
284 /*
```

```
285  * SIGCHLD handler
286  */
287 static void sigchld_handler(int signum)
288 {
289     pid_t p;
290     int status;
291     for (;;) {
292         p = waitpid(-1, &status, WUNTRACED | WNOHANG);
293         if (p == 0)
294             break;
295         explain_wait_status(p, status);
296         /* Ignore SIGCHLD of process that died from SIGTERM */
297         if (WIFEXITED(status)) {
298             /* Child has died */
299             if (head->high) {
300              highcnt--;
301             }
302             else {
303              lowcnt--;
304             }
305             printf("HIGH: %d, LOW: %d, TOTAL: %d\n", highcnt,
                ↪  lowcnt, item_count);
306             if (!highcnt) {
307              if (item_count > 1) {
308               printf("--------------BEFORE-----------------\n");
309               printf("tail->pnum: %d\n", tail->pnum);
310                  printf("tail->next->pnum: %d\n",
                    ↪  tail->next->pnum);
311                  printf("head->pnum: %d\n", head->pnum);
312                  printf("head->next->pnum: %d\n",
                    ↪  head->next->pnum);
313                  printf("head->next->next->pnum: %d\n",
                    ↪  head->next->next->pnum);
314                  node *temp = head;
315                  head = head->next;
316                  tail->next = head;
317
                    ↪  printf("--------------AFTER-----------------\n");
318                  printf("tail->pnum: %d\n", tail->pnum);
319                  printf("tail->next->pnum: %d\n",
                    ↪  tail->next->pnum);
320                  printf("head->pnum: %d\n", head->pnum);
321                  printf("head->next->pnum: %d\n",
                    ↪  head->next->pnum);
322                  printf("head->next->next->pnum: %d\n",
                    ↪  head->next->next->pnum);
```

```
323
324
325                     if(head->next != head)
326                      free(temp);
327                     --item_count;
328                     if (kill(head->pid, SIGCONT) < 0) {
329                         printf("pid that is ruinining it :%d\n",
                            ↪  head->pid);
330                         perror("sigcont error inside handler of
                            ↪  WIFEXITED");
331                         exit(1);
332                     }
333                     alarm(TQ);
334                 }
335                 else {
336                     printf("Last child has died normally\n");
337                     exit(0);
338                 }
339             }
340             else {
341              if (item_count > 1) {
342               node *temp = head;
343               head = head->next;
344               tail->next = head;
345               --item_count;
346               if(head->next != head)
347                      free(temp);
348               while (!(head->high)) {
349                head = head->next;
350                if (kill(head->pid, SIGCONT) < 0) {
351                          printf("pid that is ruinining it :%d\n",
                           ↪  head->pid);
352                          perror("sigcont error inside handler of
                           ↪  WIFEXITED");
353                          exit(1);
354                    }
355               }
356              }
357             }
358     }
359            if (WIFSTOPPED(status)) {
360                /* Child has stopped due to SIGSTOP */
361                if (!highcnt) {
362                 if (head->pid == p) {
363                  head = head->next;
364                  tail = tail->next;
```

```
365                 if (kill(head->pid, SIGCONT) < 0) {
366                     perror("sigcont error inside handler of
                        ↪  WIFSTOPPED");
367                     exit(1);
368                 }
369              }
370             }
371             else {
372              if (head->pid == p) {
373               head = head->next;
374               tail = tail->next;
375               while (!(head->high)) {
376                head = head->next;
377                tail = tail->next;
378               }
379                  if (kill(head->pid, SIGCONT) < 0) {
380                      perror("sigcont error inside handler of
                        ↪  WIFSTOPPED");
381                      exit(1);
382                  }
383              }
384             }
385                alarm(TQ);
386            }
387        }
388    }
389
390    /* Disable delivery of SIGALRM and SIGCHLD. */
391    static void signals_disable(void)
392    {
393     sigset_t sigset;
394
395     sigemptyset(&sigset);
396     sigaddset(&sigset, SIGALRM);
397     sigaddset(&sigset, SIGCHLD);
398     if (sigprocmask(SIG_BLOCK, &sigset, NULL) < 0) {
399      perror("signals_disable: sigprocmask");
400      exit(1);
401     }
402    }
403
404    /* Enable delivery of SIGALRM and SIGCHLD.  */
405    static void signals_enable(void)
406    {
407     sigset_t sigset;
408
```

```
409   sigemptyset(&sigset);
410   sigaddset(&sigset, SIGALRM);
411   sigaddset(&sigset, SIGCHLD);
412   if (sigprocmask(SIG_UNBLOCK, &sigset, NULL) < 0) {
413    perror("signals_enable: sigprocmask");
414    exit(1);
415   }
416  }
417
418
419  /* Install two signal handlers.
420   * One for SIGCHLD, one for SIGALRM.
421   * Make sure both signals are masked when one of them is running.
422   */
423  static void install_signal_handlers(void)
424  {
425   sigset_t sigset;
426   struct sigaction sa;
427   sa.sa_handler = sigchld_handler;
428   sa.sa_flags = SA_RESTART;
429   sigemptyset(&sigset);
430   sigaddset(&sigset, SIGCHLD);
431   sigaddset(&sigset, SIGALRM);
432   sa.sa_mask = sigset;
433   if (sigaction(SIGCHLD, &sa, NULL) < 0) {
434    perror("sigaction: sigchld");
435    exit(1);
436   }
437   sa.sa_handler = sigalrm_handler;
438   if (sigaction(SIGALRM, &sa, NULL) < 0) {
439    perror("sigaction: sigalrm");
440    exit(1);
441   }
442   /*
443    * Ignore SIGPIPE, so that write()s to pipes
444    * with no reader do not result in us being killed,
445    * and write() returns EPIPE instead.
446    */
447   if (signal(SIGPIPE, SIG_IGN) < 0) {
448    perror("signal: sigpipe");
449    exit(1);
450   }
451  }
452
453  static void do_shell(char *executable, int wfd, int rfd)
454  {
```

```
455   char arg1[10], arg2[10];
456   char *newargv[] = { executable, NULL, NULL, NULL };
457   char *newenviron[] = { NULL };
458   sprintf(arg1, "%05d", wfd);
459   sprintf(arg2, "%05d", rfd);
460   newargv[1] = arg1;
461   newargv[2] = arg2;
462   raise(SIGSTOP);
463   execve(executable, newargv, newenviron);
464   /* execve() only returns on error */
465   perror("scheduler: child: execve");
466   exit(1);
467  }
468
469  /* Create a new shell task.
470   *
471   * The shell gets special treatment:
472   * two pipes are created for communication and passed
473   * as command-line arguments to the executable.
474   */
475  static void sched_create_shell(char *executable, int *request_fd,
     ↪  int *return_fd)
476  {
477   pid_t p;
478   int pfds_rq[2], pfds_ret[2];
479   if (pipe(pfds_rq) < 0 || pipe(pfds_ret) < 0) {
480    perror("pipe");
481    exit(1);
482   }
483   p = fork();
484   if (p < 0) {
485    perror("scheduler: fork");
486    exit(1);
487   }
488   if (p == 0) {
489    /* Child */
490    close(pfds_rq[0]);
491    close(pfds_ret[1]);
492    change_pname(executable);
493    do_shell(executable, pfds_rq[1], pfds_ret[0]);
494    assert(0);
495   }
496   /* Parent */
497   if(head->pnum == 0)
498    head->pid = p;
499   close(pfds_rq[1]);
```

```
500   close(pfds_ret[0]);
501   *request_fd = pfds_rq[0];
502   *return_fd = pfds_ret[1];
503  }
504
505  static void shell_request_loop(int request_fd, int return_fd)
506  {
507   int ret;
508   struct request_struct rq;
509
510   /*
511    * Keep receiving requests from the shell.
512    */
513   for (;;) {
514    if (read(request_fd, &rq, sizeof(rq)) != sizeof(rq)) {
515     perror("scheduler: read from shell");
516     fprintf(stderr, "Scheduler: giving up on shell request
       ↪  processing.\n");
517     break;
518    }
519    printf("read from pipe and disabling signals\n");
520    signals_disable();
521    ret = process_request(&rq);
522    signals_enable();
523
524    if (write(return_fd, &ret, sizeof(ret)) != sizeof(ret)) {
525     perror("scheduler: write to shell");
526     fprintf(stderr, "Scheduler: giving up on shell request
       ↪  processing.\n");
527     break;
528    }
529   }
530  }
531
532  int main(int argc, char *argv[])
533  {
534   int nproc;
535   pid_t p;
536   /* Two file descriptors for communication with the shell */
537   static int request_fd, return_fd;
538
539   head = safe_malloc(sizeof(node));
540   tail = head;
541
542   head->pnum = 0;
543   head->name = SHELL_EXECUTABLE_NAME;
```

```
544   head->next = NULL;
545   item_count++;
546   /* Create the shell. */
547   sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd,
      ↪  &return_fd);
548   /*
549    * For each of argv[1] to argv[argc - 1],
550    * create a new child process, add it to the process list.
551    */
552   nproc = argc; /* number of proccesses goes here */
553      for (int i = 1; i < nproc; i++) {
554          p = fork();
555          if (p < 0) {
556              /* fork failed */
557              perror("fork");
558              exit(1);
559          }
560          if (p == 0) {
561              /* In child process */
562              char executable[TASK_NAME_SZ];
563              strcpy(executable, argv[i]);
564  //          printf("%s\n", executable);
565              change_pname(executable);
566              char *newargv[] = { executable, NULL, NULL, NULL };
567              char *newenviron[] = { NULL };
568              kill(getpid(), SIGSTOP);
569              execve(executable, newargv, newenviron);
570          }
571          else {
572              insertq(p, argv[i], i);
573          }
574      }
575      total = item_count;
576      lowcnt = item_count;
577
578      /* Circularize the list */
579      tail->next = head;
580
581   /* Wait for all children to raise SIGSTOP before exec()ing. */
582   wait_for_ready_children(nproc);
583
584   // show_pstree(getpid());
585
586   /* Install SIGALRM and SIGCHLD handlers. */
587   install_signal_handlers();
588
```

```
589   if (nproc == 0) {
590    fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
591    exit(1);
592   }
593
594
595   if (kill(head->pid, SIGCONT) < 0) {
596           perror("sigcont error");
597           exit(1);
598      }
599
600      alarm(TQ);
601
602
603   shell_request_loop(request_fd, return_fd);
604   if (kill(head->pid, SIGCONT) < 0) {
605       perror("sigcont error");
606       exit(1);
607      }
608   alarm(TQ);
609   /* Now that the shell is gone, just loop forever
610    * until we exit from inside a signal handler.
611    */
612   while (pause())
613    ;
614
615   /* Unreachable */
616   fprintf(stderr, "Internal error: Reached unreachable point\n");
617   return 1;
618  }
```

3.2 Έξοδος

```
My PID = 15907: Child PID = 15908 has been stopped by a signal, signo = 19
My PID = 15907: Child PID = 15909 has been stopped by a signal, signo = 19
My PID = 15907: Child PID = 15910 has been stopped by a signal, signo = 19

This is the Shell. Welcome.

Shell> p
Shell: issuing request...
Shell: receiving request return value...
read from pipe and disabling signals
--- PRINTING PROCESS LIST: ---
CURRENT RUNNING PROCESS:***** PNUM=0, PID=15908 ,P_EX_NAME=shell, PRIORITY:LOW *****
                        ***** PNUM=1, PID=15909 ,P_EX_NAME=prog, PRIORITY:LOW *****
```

```
                        ***** PNUM=2, PID=15910 ,P_EX_NAME=prog, PRIORITY:LOW *****
Shell> My PID = 15907: Child PID = 15908 has been stopped by a signal, signo = 19
prog: Starting, NMSG = 200, delay = 152
prog[15909]: This is message 0
prog[15909]: This is message 1
prog[15909]: This is message 2
prog[15909]: This is message 3
prog[15909]: This is message 4
prog[15909]: This is message 5
prog[15909]: This is message 6
prog[15909]: This is message 7
prog[15909]: This is message 8
My PID = 15907: Child PID = 15909 has been stopped by a signal, signo = 19
prog: Starting, NMSG = 200, delay = 68
prog[15910]: This is message 0
prog[15910]: This is message 1
prog[15910]: This is message 2
prog[15910]: This is message 3
prog[15910]: This is message 4
prog[15910]: This is message 5
prog[15910]: This is message 6
prog[15910]: This is message 7
prog[15910]: This is message 8
prog[15910]: This is message 9
prog[15910]: This is message 10
prog[15910]: This is message 11
prog[15910]: This is message 12
prog[15910]: This is message 13
prog[15910]: This is message 14
prog[15910]: This is message 15
prog[15910]: This is message 16
prog[15910]: This is message 17
prog[15910]: This is message 18
My PID = 15907: Child PID = 15910 has been stopped by a signal, signo = 19
```

3.3 Ερώτημα 1

Ένα τέτοιο σενάριο δημιουργείται αν υπάρχει στην ουρά διεργασιών τουλάχιστον μια "high" διεργασία συνεχούς λειτουργίας και ο φλοιός γίνει "low". Στην περίπτωση αυτή η εν λόγω διεργασία θα δρομολογείται πάντοτε με προτεραιότητα έναντι του φλοιού. Αυτό έχει ως αποτέλεσμα ο φλοιός να μην ξαναεκτελεστεί και το σχήμα λειτουργίας χρονοδρομολογητή-φλοιού που έχουμε υπόψιν μας να καταστρατηγηθεί. Ο μόνος τρόπος επαναφοράς της φυσιολογικής λειτουργίας είναι με εξωγενή τερματισμό της ή των διεργασιών "high", ώστε να δοθεί η ευκαιρία στον φλοιό να συνεχίσει.