Λειτουργικά Συστήματα, 6^{ο} Εξάμηνο ΗΜΜΥ

Ακαδημαϊκή Περίοδος 2019-2020

Άσκηση 4: Χρονοδρομολόγηση

Ομάδα: oslabb33

Γιαννούλης Παναγιώτης (03117812)

Κανελλόπουλος Σωτήρης (03117101)

1.1 Αρχικά παραθέτουμε το struct ουράς που φτιάξαμε για αυτό και το επόμενο ερώτημα. Ο υπόλοιπος κώδικας που γράφτηκε για την άσκηση υπάρχει ως αρχείο στο συμπιεσμένο που κατατέθηκε (δεν παρατίθεται εδώ ως εικόνα λόγω μεγέθους).

```
typedef struct Node_q {
  pid_t pid;
  int id;
  char *name;
  struct Node_q *next;
} Node;


typedef struct queue {
  int count;
  int max;
  Node *front;
  Node *rear;
} queue;


void init_queue(queue *q);

void enqueue(queue *q, pid_t pid, const char *name);

void delete_node(queue *q, Node *n);

void dequeue(queue *q, pid_t pid);

void move_to_end(queue *q);

void display_queue(Node *head);
```

Παραθέτουμε ενδεικτικά κάποια τμήματα της εξόδου του προγράμματός μας. Στην πρώτη εικόνα βλέπουμε ότι η διεργασία με id 0 τερματίζεται και έτσι στο επόμενο βήμα υπάρχουν στην ουρά μόνο οι διεργασίες 1, 2, 3. Στη δεύτερη εικόνα βλέπουμε ότι η διαδικασία συνεχίζεται κυκλικά, δηλαδή τώρα η ουρά είναι 2, 3, 1. Στην τρίτη εικόνα βλέπουμε την ολοκλήρωση του προγράμματος, όταν η ουρά έχει αδειάσει.

```
My PID = 29647: Child PID = 29648 terminated normally, exit status = 0
pid = 29649     name = prog with id = 1
pid = 29650     name = prog with id = 2
pid = 29651     name = prog with id = 3
we have 3 remaining processes
Parent: Received SIGCHLD, child is dead. Exiting.
prog[29649]: This is message 13
prog[29649]: This is message 14
prog[29649]: This is message 15
prog[29649]: This is message 16
prog[29649]: This is message 17
prog[29649]: This is message 18
prog[29649]: This is message 19
prog[29649]: This is message 20
prog[29649]: This is message 21
prog[29649]: This is message 22
prog[29649]: This is message 23
prog[29649]: This is message 24
ALARM! 2 seconds have passed.
pid = 29650     name = prog with id = 2
pid = 29651     name = prog with id = 3
pid = 29649     name = prog with id = 1
My PID = 29647: Child PID = 29649 has been stopped by a signal, signo = 19
Parent: Child has been stopped. Moving right along...
```

```
My PID = 29647: Child PID = 29650 terminated normally, exit status = 0
pid = 29651     name = prog with id = 3
we have 1 remaining processes
Parent: Received SIGCHLD, child is dead. Exiting.
prog[29651]: This is message 20
prog[29651]: This is message 21
prog[29651]: This is message 22
prog[29651]: This is message 23
prog[29651]: This is message 24
prog[29651]: This is message 25
prog[29651]: This is message 26
prog[29651]: This is message 27
prog[29651]: This is message 28
ALARM! 2 seconds have passed.
pid = 29651     name = prog with id = 3
My PID = 29647: Child PID = 29651 has been stopped by a signal, signo = 19
Parent: Child has been stopped. Moving right along...
prog[29651]: This is message 29
My PID = 29647: Child PID = 29651 terminated normally, exit status = 0
You are the mastermind, you did it.
Bye
```

Απαντήσεις στις ερωτήσεις

1. Κατά τη διάρκεια εκτέλεσης ενός handler για ένα συγκεκριμένο σήμα, όλα τα σήματα ίδου τύπου μπλοκάρονται και περιμένουν ώστε να τα χειριστεί ο handler αμέσως μετά την ολοκλήρωση του τρέχοντος χειρισμού. Όμως, σήματα διαφορετικού είδους μπορούν να διακόψουν κανονικά τη λειτουργία του handler. Γι' αυτό το λόγο στην υλοποίηση εγκαθιστούμε αρχικά τους handlers για τα σήματα SIGCHLD και SIGALRM με τη χρήση ενός struct τύπου sigaction. Έπειτα μέσω ενός πεδίου sa_mask ορίζουμε τα σήματα να μπλοκάρονται και να χειρίζονται αργότερα ανεξάρτητα από το αν θα είναι ίδιου τύπου με το σήμα που χειρίζεται εκείνη τη στιγμή ένας από τους hnadlers που έχουμε εισάγει. Επομένως, σε περίπτωση που εκτελείται η συνάρτηση χειρισμού του σήματος SIGCHLD και έρθει σήμα SIGALRM ή το αντίστροφο επιτυγχάνουμε να μπλοκάρεται το σήμα αυτό και να περιμένει ώστε να χειριστεί εν καιρώ. Ένας πραγματικός χρονοδρομολογητής σε χώρο πυρήνα ενημερώνεται για τις αλλαγές των διεργασιών μέσω hardware interrupts (διακοπτών) αντί για σήματα.

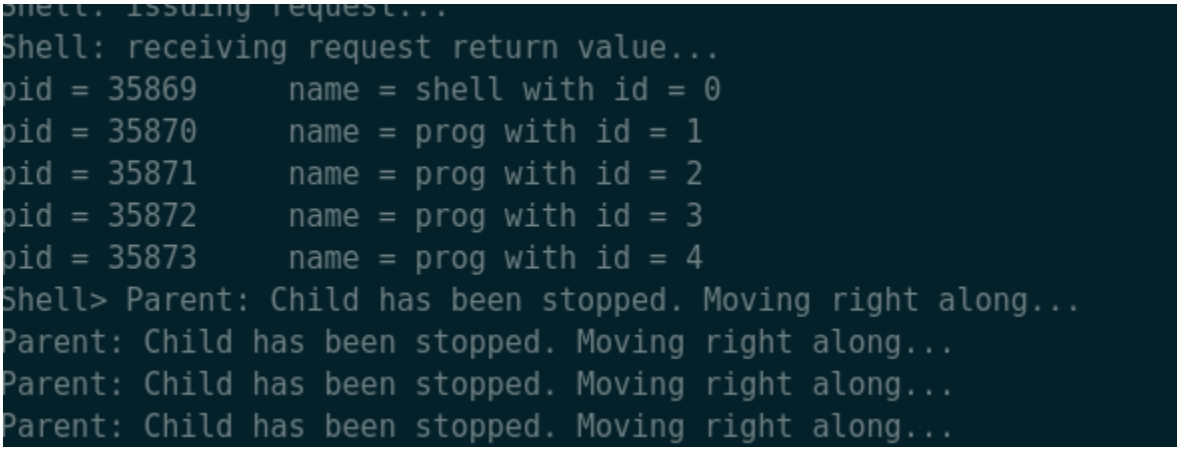
2. Διαισθητικά θα περιμέναμε ότι το σήμα SIGCHLD αναφέρεται πάντα στην κεφαλή της ουράς, αφού αυτή βρίσκεται υπό εκτέλεση. Ωστόσο, στην πραγματικότητα το σήμα SIGCHLD μπορεί να αφορά οποιοδήποτε στοιχείο της ουράς, καθώς μπορεί να έχει τερματιστεί κάποια διεργασία από εξωτερικό παράγοντα. Για αυτό το λόγο, στον κώδικά μας έχουμε διαχωρίσει τις περιπτώσεις WIFEXITED (η οποία πάντα αφορά την κεφαλή τους ουράς, καθώς αυτή βρίσκεται υπό εκτέλεση και επομένως μόνο αυτή μπορεί να τερματιστεί από μόνη της) και WIFSIGNALED (η οποία μπορεί να αναφέρεται σε οποιαδήποτε διεργασία της ουράς, σε περίπτωση που αυτή τερματίστηκε από σήμα), καθώς στη δεύτερη πρέπει να ξέρουμε για ποια διεργασία ήρθε το σήμα.

3. Το SIGCHLD χρησιμεύει στην επιβεβαίωση παραλαβής του σήματος SIGSTOP και επομένως στην επιβεβαίωση ότι η διεργασία τερματίστηκε κανονικά. Αν δεν είχαμε αυτή την επιβεβαίωση, τότε σε περίπτωση που το σήμα δεν λαμβανόταν θα είχαμε ενδεχομένως λανθασμένη λειτουργία του χρονοδρομολογητή (π.χ. να τρέχουν δύο διεργασίες ταυτόχρονα ή να τερματιστεί κάποια χωρίς να τρέξει η επόμενη). Επίσης, αν τερματιστεί ομαλά ένα παιδί ή τερματιστεί από κάποιον εξωτερικό παράγοντα, τότε χρειαζόμαστε τον SIGCHLD_HANDLER για να το βγάλουμε από την ουρά.

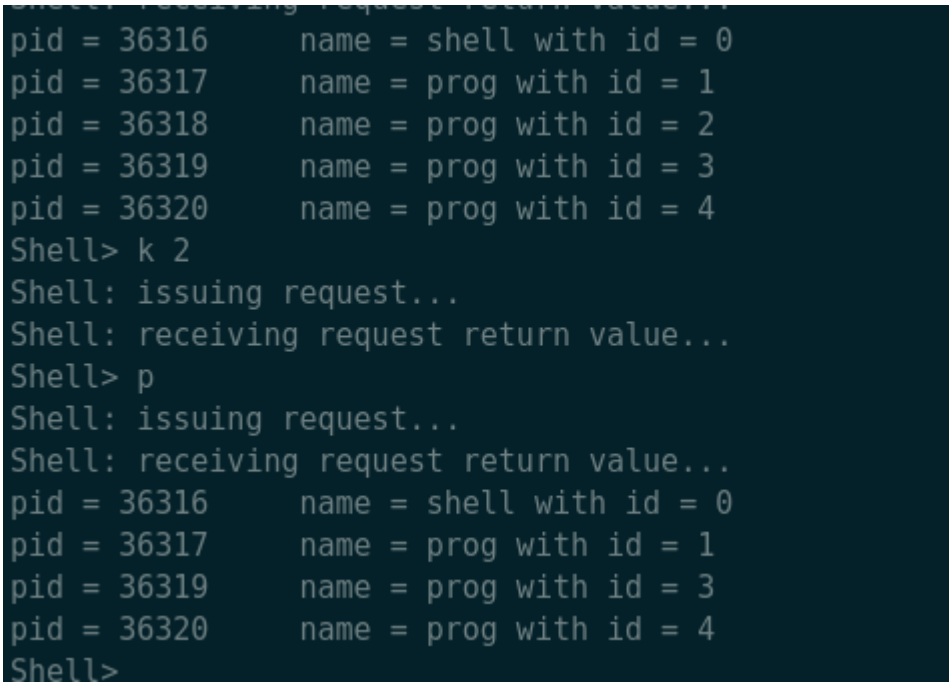
1.2 Παραθέτουμε ενδεικτικά μερικά τμήματα της εξόδου του προγράμματός μας:

```
Shell> p
Shell: issuing request...
Shell: receiving request return value...
pid = 35994     name = shell with id = 0
pid = 35995     name = prog with id = 1
pid = 35996     name = prog with id = 2
pid = 35997     name = prog with id = 3
pid = 35998     name = prog with id = 4
Shell>
```

Εδώ βλέπουμε την αρχική έξοδο του προγράμματος, αν του ζητήσουμε με την εντολή p στο shell να μας τυπώσει την ουρά.



Εδώ βλέπουμε τι συμβαίνει καθώς το πρόγραμμα συνεχίζει, χωρίς κάποια παρέμβαση από μας μέσω του shell. Στο κάτω μέρος της εικόνας βλέπουμε ότι οι διεργασίες σταματούν προσωρινά μέσω του alarm, όπως στην πρώτη άσκηση.



Εδώ χρησιμοποιήσαμε την εντολή k για να τερματίσουμε τη διεργασία με id = 2. Έτσι, τυπώνοντας την ουρά βλέπουμε ότι τώρα περιέχει μόνο τις διεργασίες 0, 1, 3, 4.

```
Shell: receiving request return value...
pid = 36316     name = shell with id = 0
pid = 36317     name = prog with id = 1
pid = 36319     name = prog with id = 3
pid = 36320     name = prog with id = 4
Shell> e prog
Shell: issuing request...
Shell: receiving request return value...
Shell> p
Shell: issuing request...
Shell: receiving request return value...
pid = 36316     name = shell with id = 0
pid = 36317     name = prog with id = 1
pid = 36319     name = prog with id = 3
pid = 36320     name = prog with id = 4
pid = 36359     name = prog with id = 5
Shell>
```

Εδώ με την εντολή e προσθέσαμε μια νέα διεργασία prog στην ουρά, η οποία παρακάτω φαίνεται ότι έχει id = 5.

```
Shell: issuing request...
Shell: receiving request return value...
pid = 36316     name = shell with id = 0
Shell> q
Shell: Exiting. Goodbye.
You are the mastermind, you did it.
Bye
```

Σε αυτή την εικόνα βλέπουμε τη λήξη της εκτέλεσης του προγράμματος, όταν έχουν ολοκληρωθεί όλες οι διεργασίες πλην του shell, το οποίο τελικά τερματίζουμε χειροκίνητα με την εντολή q.

Απαντήσεις στις ερωτήσεις

1. Ως τρέχουσα διεργασία εμφανίζεται πάντα το shell (id = 0), το οποίο είναι λογικό, αφού η εντολή p δεν εκτελείται όταν την εισάγει ο χρήστης, αλλά όταν έρθει η σειρά του shell να τρέξει. Αυτό θα μπορούσε να μη συμβαίνει αν είχαμε διαφορετική υλοποίηση, π.χ. αν δεν χρονοδρομολογούνταν ο φλοιός.

2. Αν ερχόταν κάποιο σήμα κατά τη διάρκεια της εκτέλεσης αιτήσεων του φλοιού, τότε θα μεταφερόμασταν αυτόματα στον handler του αντίστοιχου σήματος. Αυτό ενδεχομένως θα

δημιουργούσε προβλήματα, καθώς οι εντολές του φλοιού μεταβάλουν την ουρά (π.χ. διαγράφουν και προσθέτουν στοιχεία σε αυτή). Για παράδειγμα, αν βρισκόμασταν στη μέση εξυπηρέτησης εντολής k 2 θα μπορούσε οποιοδήποτε σήμα να διακόψει απότομα τη λειτουργία αυτή κυριολεκτικά σε μια τυχαία εντολή και προφανώς το πιο πιθανό θα ήταν να καταστρεφόταν η δομή μας. Στην καλύτερη περίπτωση μάλλον λίγο αργότερα θα προέκυπτε κάποιο segmentation fault. Γι' αυτό το λόγο κάνουμε disable τα σήματα όταν εξυπηρετούμε αιτήσεις του φλοιού.

1.3 Τροποποιήσαμε το struct μας ώστε να υποστηρίζει τις νέες απαραίτητες λειτουργίες:

```
typedef struct Node_q {
  pid_t pid;
  int id;
  char *name;
  struct Node_q *next;
  char priority;
} Node;


typedef struct queue {
  int count;
  int max;
  Node *front;
  Node *rear;
  Node *rear_high;
} queue;

void init_queue(queue *q);

void enqueue(queue *q, pid_t pid, const char *name);

void delete_node(queue *q, Node *n);

void dequeue(queue *q, pid_t pid);

void move_to_end(queue *q);

void change_to_low (queue *q, Node *n);

void change_to_high (queue *q, Node* n);

void renew_rear_high (queue *q);

void display_queue(Node *head);
```

Όπως θα παρατηρήσουμε και στη συνέχεια, σχεδιάσαμε τον χρονοδρομολογητή μας κατά την αλλαγή του priority μιας διεργασίας από high σε low και το αντίστροφο, να μεταφέρει τη διεργασία αυτή στην τελευταία θέση των διεργασιών με ίδια προτεραιότητα. Παραθέτουμε ξανά κάποια τμήματα της εξόδου του προγράμματός μας:

```
Shell> h 0
prog[5314]: This is message 18
prog[5314]: This is message 19
prog[5314]: This is message 20
```

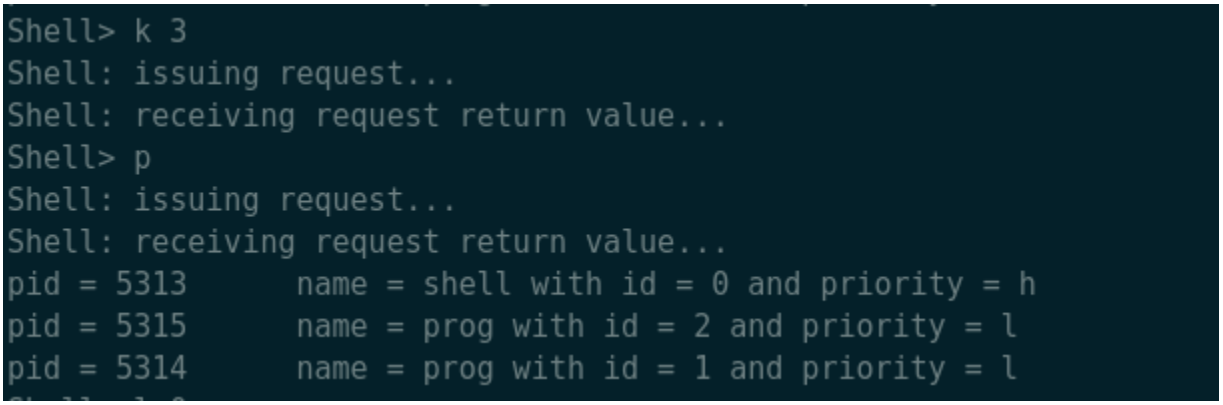
Εδώ δίνουμε υψηλή προτεραιότητα στο shell με την εντολή h 0.

```
Shell> p
Shell: issuing request...
Shell: receiving request return value...
pid = 5313      name = shell with id = 0 and priority = h
pid = 5314      name = prog with id = 1 and priority = l
pid = 5315      name = prog with id = 2 and priority = l
pid = 5316      name = prog with id = 3 and priority = l
```

Παρατηρούμε ότι τυπώνοντας την ουρά με την εντολή p, πράγματι όλες οι διεργασίες εκτός του shell έχουν χαμηλή προτεραιότητα.

```
Shell> h 1
Shell: issuing request...
Shell: receiving request return value...
Shell> prog[5314]: This is message 26
prog[5314]: This is message 27
prog[5314]: This is message 28
prog[5314]: This is message 29
prog[5314]: This is message 30
```

Δίνουμε υψηλή προτεραιότητα και στη διεργασία με id = 1. Παρατηρούμε από τα μηνύματα prog[5314] ότι πράγματι τρέχει η διεργασία αυτή.



Ομοίως με τη δεύτερη άσκηση, αφαιρούμε από την ουρά μία από τις διεργασίες χρησιμοποιώντας την εντολή k 3. Σημειώνουμε ότι εδώ έχει μεσολαβήσει και μια εντολή l 1, η οποία δεν φαίνεται στο screenshot, για αυτό και βλέπουμε ότι η διεργασία με id = 1 έχει τοποθετηθεί στο τέλος των low διεργασιών.

Δίνουμε ξανά χαμηλή προτεραιότητα στο shell. Σε αυτό το σημείο φαίνεται να τρέχει η διεργασία με pid = 5315 (id = 2), η οποία με βάση τα παραπάνω ήταν η πρώτη low διεργασία στην ουρά. Βλέπουμε λοιπόν ότι αφού δεν υπάρχει high διεργασία τώρα τρέχουν και οι low.

```
Shell> h 0
Shell: issuing request...
Shell: receiving request return value...
Shell> e prog
Shell: issuing request...
Shell: receiving request return value...
Shell> p
Shell: issuing request...
Shell: receiving request return value...
pid = 5313      name = shell with id = 0 and priority = h
pid = 5315      name = prog with id = 2 and priority = l
pid = 5367      name = prog with id = 4 and priority = l
pid = 5369      name = prog with id = 5 and priority = l
Shell>
```

Με τη χρήση της εντολής e prog προσθέτουμε μια νέα διεργασία με id = 5 στην ουρά, η οποία αρχικά έχει χαμηλή προτεραιότητα, όπως φαίνεται παραπάνω.

```
Shell> p
Shell: issuing request...
Shell: receiving request return value...
pid = 5313      name = shell with id = 0 and priority = l
Shell> q
Shell: Exiting. Goodbye.
You are the mastermind, you did it.
Bye
```

Σε αυτό το σημείο βλέπουμε ότι η μόνη διεργασία που έχει μείνει στην ουρά είναι το shell, με χαμηλή προτεραιότητα. Σημειώνουμε ότι αφού δεν υπάρχουν διεργασίες με υψηλή προτεραιότητα, το shell τρέχει παρόλο που έχει χαμηλή. Έτσι, με την εντολή q το shell τερματίζεται και το πρόγραμμα ολοκληρώνεται.

<u>Απαντήσεις στις ερωτήσεις:</u>

1. Αν έχουμε μια διεργασία A με high priority, η οποία για κάποιο λόγο δεν τερματίζει (π.χ. από τη φύση της είναι ατέρμονη ή περιμένει κάποια συνθήκη για να τερματίσει, η οποία δε συμβαίνει ποτέ) και κάνουμε το shell low priority με την εντολή l 0, τότε οι low διεργασίες δεν θα τρέξουν ποτέ ανεξαρτήτως των εισόδων που βάζουμε στο shell. Αυτό συμβαίνει καθώς η A θα τρέχει ατέρμονα, χωρίς να έρχεται ποτέ η σειρά των low διεργασιών και δεν έχουμε τρόπο να κάνουμε την A low, καθώς το shell είναι επίσης low και δε θα τρέξει ποτέ. Επομένως, σε αυτό το σενάριο έχουμε λιμοκτονία.