ΕΘΝΙΚΟ ΜΕΤΣΟΒΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ: ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΜΑΘΗΜΑ: Λειτουργικά Συστήματα

Λειτουργικά Συστήματα

ΟΝΟΜΑΤΕΠΩΝΥΜΟ ΜΕΛΩΝ ΟΜΑΔΑΣ :
ΣΤΑΜΑΤΗΣ ΑΛΕΞΑΝΔΡΟΠΟΥΛΟΣ (03117060)
ΓΚΟΤΣΗ ΠΟΛΥΤΙΜΗ – ΑΝΝΑ (03117201)

Ομάδα: (oslaba16)

ΑΘΗΝΑ
2020

3.1 Άσκηση 1.1

Στην άσκηση αυτή υλοποιήσαμε ένα ασύγχρονο χρονοδρομολογητή κυκλικής επαναφοράς (round-robin) (με ουρά έτοιμων διεργασιών FIFO) βασισμένο σε σήματα. Αυτός εκτελείται ως γονική διεργασία, στο χώρο χρήστη , μοιράζοντας τον υπολογιστικό χρόνο σε διεργασίες-παιδιά. Προκειμένου να ελέγχει τις διεργασίες ο χρονοδρομολογητής χρησιμοποιεί τα σήματα SIGSTOP και SIGCONT για τη διακοπή και την ενεργοποίηση κάθε διεργασίας, αντίστοιχα. Κάθε διεργασία εκτελείται για χρονικό διάστημα το πολύ ίσο με το κβάντο χρόνου tq. Αν τερματιστεί πριν από το τέλος του κβάντου χρόνου, ο χρονοδρομολογητής την αφαιρεί από την ουρά των έτοιμων διεργασιών και ενεργοποιεί την επόμενη. Αν το κβάντο χρόνου εκπνεύσει χωρίς η διεργασία να έχει ολοκληρώσει την εκτέλεσή της, τότε αυτή διακόπτεται, τοποθετείται στο τέλος της ουράς έτοιμων διεργασιών και ενεργοποιείται η επόμενη.

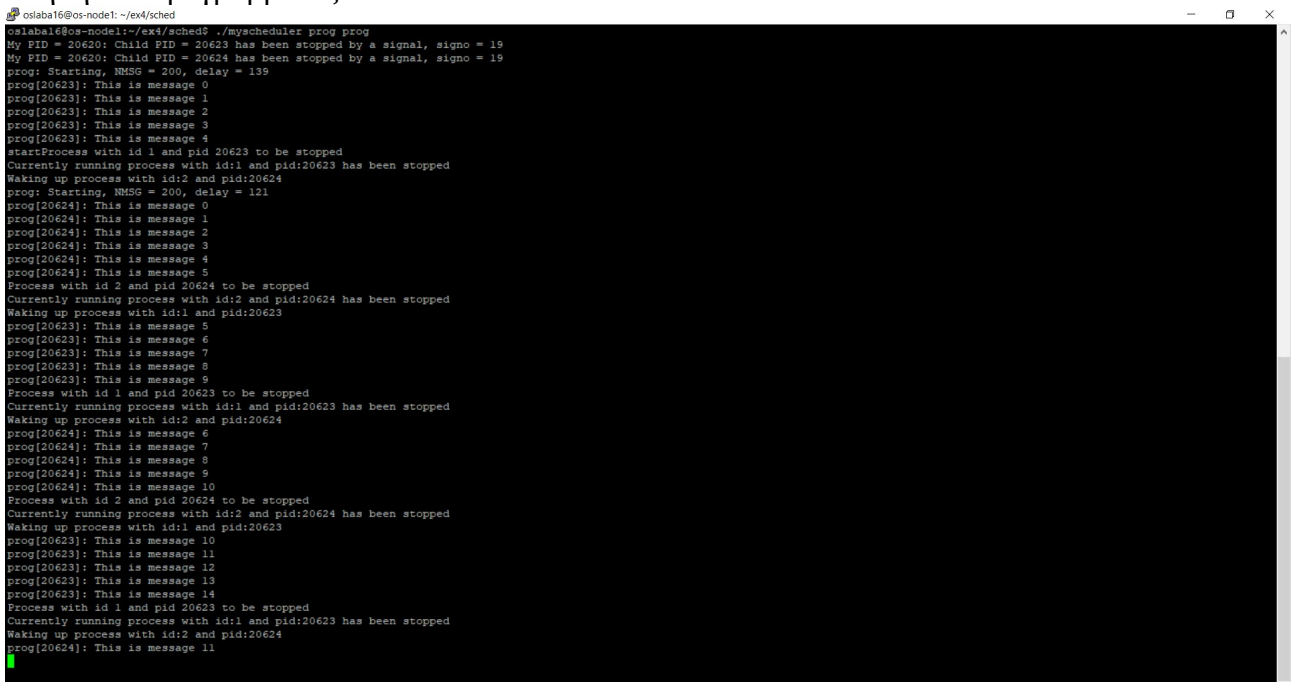
Εφόσον η χρονοδρομολόγηση είναι κυκλική, και οι διεργασίες χρονοδρομολογούνται με την σειρά για ίσα κβάντα χρόνου, επιλέξαμε για την υλοποίηση της ουράς έτοιμων διεργασιών μία κυκλική λίστα, κάθε κόμβος της οποίας δείχνει σε μια διεργασία (στο pcb αυτής) (Το pcb (process control block) της διεργασίας αποτελείται από το id, το pid και το όνομα αυτής, όπου το id της διεργασίας αποτελεί έναν αύξοντα αριθμό με βάση το σε ποια σειρά δημιουργήθηκε (χρησιμοποιείται η μεταβλήτή number_of_processes για τον σκοπό αυτό στην υλοποίησή μας). Η υλοποίηση αυτή ταιριάζει με την μορφή χρονοδρομολόγησης, στην οποία δεν υπάρχει αρχή και τέλος αλλά απλή κυκλική επιλογή της τρέχουσας διεργασίας. Έτσι, ο χρονοδορμολογητής διατηρεί για την αναφορά στην κυκλική λίστα έναν δείκτη στον κόμβο της λίστας που αντιστοιχεί στην running διεργασία. Κάθε φορά που το κβάντο χρόνου της τρέχουσας διεργασίας εκπνέει, ο δείκτης αυτός προχωράει στην επόμενη διεργασία και άρα η διεργασία που σταματήθηκε βρίσκεται ουσιαστικά πλέον στο τέλος της ουράς. Παρατηρούμε ότι με την υλοποίηση αυτή αποφεύγονται περιττές ενέργειας που θα ήταν απαραίτητες αν η λίστα δεν ήταν κυκλική, όπως η μετακίνηση της διεργασίας που σταματήθηκε από τον χρονοδρομολογητή κάθε φορά από την αρχή στο τέλος της ουράς.

Όταν τρέχουμε τον χρονοδρομολογητή, αυτός δημιουργεί με την σειρά τις διεργασίες που δώσαμε ως ορίσματα, κάνοντας fork(), και στη συνέχεια περιμένει να θέσουν σε σταματήσουν την λειτουργία τους με το σήμα SIGSTOP και ύστερα θέτει τους signal handlers για να λαμβάνει τα σήματα SIGALRM και SIGCHILD και θέτει ως running την πρώτη διεργασία στην λίστα. Στη συνέχεια ανάλογα με τα σήματα που λαμβάνει κάθε φορά χρησιμοποιεί τους κατάλληλους χειριστές για να χρονοδρομολογεί τις διεργασίες και να διατηρεί την κυκλική λίστα διεργασιών ενημερωμένη.

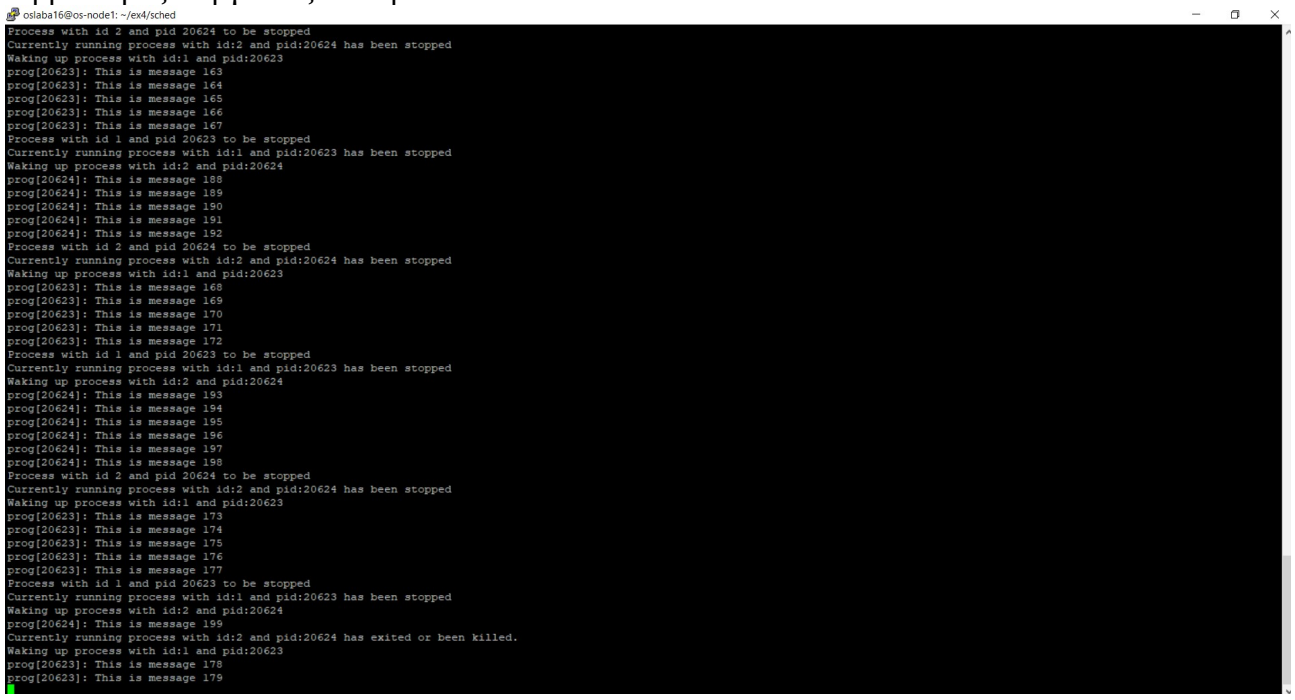
Ο κώδικας της άσκησης παρατίθεται στο τέλος της αναφοράς και στο αντίστοιχο αρχείο.

Ενδεικτικά παρατίθενται τμήματα της εξόδου του προγράμματος:

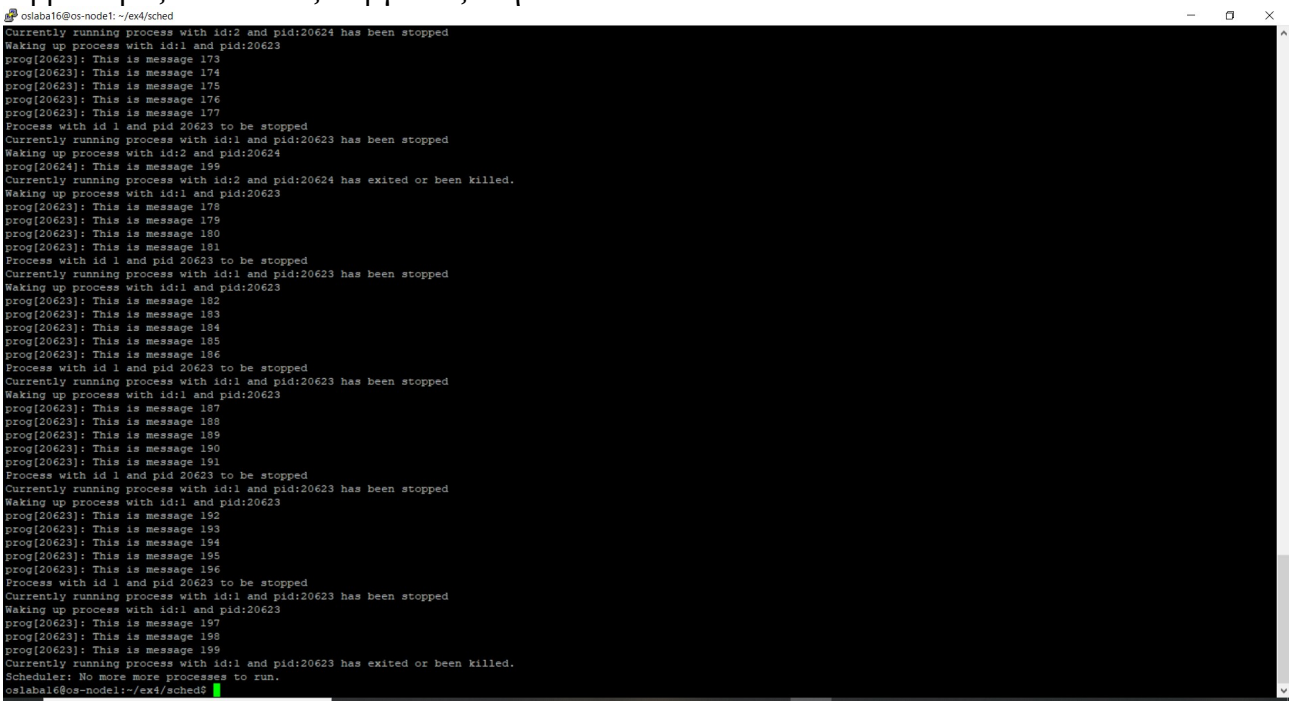
Κλήση του προγράμματος:



Τερματισμός διεργασίας ενδιάμεσα:



Τερματισμός τελευταίας διεργασίας στην λίστα:



Ερωτήσεις:

1. Παρατηρούμε ότι στην συνάρτηση install_signal_handlers() ορίζεται μία μάσκα για τα σήματα SIGSTOP, SIGALARM. Και επομένως όταν εκτελείται κάποιος από τους signal handlers "sigchld_handler()", "sigalrm_handler()" τα σήματα μπλοκάρονται λόγω της μάσκας θα εκτελεστεί ο χειριστής του σήματος SIGALARM ενώ εκτελείται η συνάρτηση χειρισμού του SIGCHLD ή το αντίστροφο.
Ο χρονοδρομολογητής σε χώρο πυρήνα αντιμετωπίζει ανάλογα ενδεχόμενα με διαφορετικό τρόπο. Μια διεργασία εκτελείται σε χώρο χρήστη, ενώ ο χρονοδρομολογητής είναι χειριζόμενος από το λειτουργικό σύστημα και εκτελείται σε χώρο πυρήνα, αντιλαμβανόμενος system calls και hardware interrupts. Λαμβάνοντας αυτά το λειτουργικό σύστημα, μπορεί ο scheduler να πρέπει να δράσει άμεσα (π.χ. σε περιπτώσεις διακοπών όπως η άνοδος της θερμοκρασίας του επεξεργαστή ή σε περίπτωση που η τρέχουσα διεργασία βγήκε εκτός των ορίων της μνήμης της (segmentation fault) και όχι να περιμένει να ολοκληρωθεί ο όποιος χειριστής εκτελούταν εκείνη την ώρα, και επομένως να περάσει από τον χώρο χρήστη της διεργασίας σε χώρο πυρήνα, να προσδιορίσει το είδος της διακοπής ή του system call και να δράσει ανάλογα, επιστρέφοντας τον έλεγχο στην διεργασία και στέλνοντας κάποιο σήμα σε αυτή ή κάνοντας παύση ή kill στην διεργασία.

2. Το σήμα SIGCHILD αναφέρεται κάθε φορά σε οποιαδήποτε διεργασία τερμάτισε (με exit()) ή τερματίστηκε (με kill()). Έτσι δεν αναμένουμε να αναφέρεται σε κάποια συγκεκριμένη διεργασία το σήμα SIGCHILD, αλλά σε οποιαδήποτε από τις διεργασίες παιδιά του χρονοδρομολογητή. Παρατηρούμε ότι πληκτρολογώντας την εντολή: kill -15 <pid>, όπου το πεδίο <pid> συμπληρώνεται με το pid της διεργασίας παιδιού του χρονοδρομολογητή που θέλουμε να σκοτώσουμε, σε ένα δεύτερο τερματικό, προκύπτει το error message: No such process , επομένως δεν μας είναι εφικτό να στείλουμε κάποιο σήμα τερματισμού μίας child process. Αν ωστόσο στελνόταν κάποιο τέτοιο σήμα, αναμένουμε πως ο χρονοδρομολογητής θα λάμβανε σήμα SIGCHLD και κάνοντας waitpid, explain_wait_status, θα μάθαινε ότι η διεργασία παιδί του πέθανε και θα το χειριζόταν κατάλληλα, αφαιρώντας την από την λίστα των έτοιμων διεργασιών που χρονοδρομολογεί.

3. Απαιτούνται δύο σήματα για την υλοποίηση του χρονοδρομολογητή με σωστό συγχρονισμό για τον εξής λόγο: Σε περίπτωση που το κβάντο χρόνου λήξει, θα σταλεί στην τρέχουσα διεργασία ένα σήμα SIGSTOP. Η αναμενόμενη λειτουργία θα ήταν η διεργασία να αναστείλει την λειτουργία της και ο χρονοδρομολογητής να ξυπνήσει την επόμενη. Αν ωστόσο δεν αναμένει κάποιο σήμα SIGCHILD για να βεβαιωθεί ότι η προηγούμενη διεργασία έχει αναστείλει την λειτουργία της, τότε σε περίπτωση που υπάρξει κάποια καθυστέρηση της λήψης του σήματος και υλοποίησης του handler του από την διεργασία παιδί, ώστε να αναστείλει την λειτουργία της, μπορεί να προκύψει η ανεπιθύμητη κατάσταση κατά την οποία ο ενεργοποιείται η επόμενη διερεγασία πριν ανασταλθεί η επόμενη, και δύο διεργασίες είναι running ταυτόχρονα. Έτσι, χρησιμοποιούμε δύο διαφορετικά σήματα, ώστε να είμαστε πάντα σίγουροι για την σωστή σειρά εκτέλεσης των απαραίτητων ενεργειών από την διεργασία παιδί και τον χρονοδρομολογητή.
Τα παραπάνω αφορούν την περίπτωση που δεν χρησιμοποιούμε το σήμα SIGCHLD για την εναλλαγή διεργασίας κατά την λήξη του κβάντου χρόνου. Αν τώρα θεωρήσουμε ότι δεν χρησιμοποιείται γενικά χειριστής για το σήμα SIGCHLD κατά την υλοποίηση του χρονοδρομολογητή, εκτός του παραπάνω προβλήματος, δημιουργούνται και άλλες προβληματικές καταστάσεις αφού στην περίπτωση αυτή όταν μια διεργασία παιδί τερματίζει είτε ολοκληρώνοντας την λειτουργία της είτε βίαια ο χρονοδρομολογητής-πατέρας δεν κάνει wait() και δεν μαθαίνει για τον τερματισμό της, με αποτέλεσμα να την διατηρεί ως έτοιμη διεργασία στην λίστα έτοιμων διεργασίων και να προσπαθεί να την χρονοδρομολογήσει μαζί με τις υπόλοιπες.

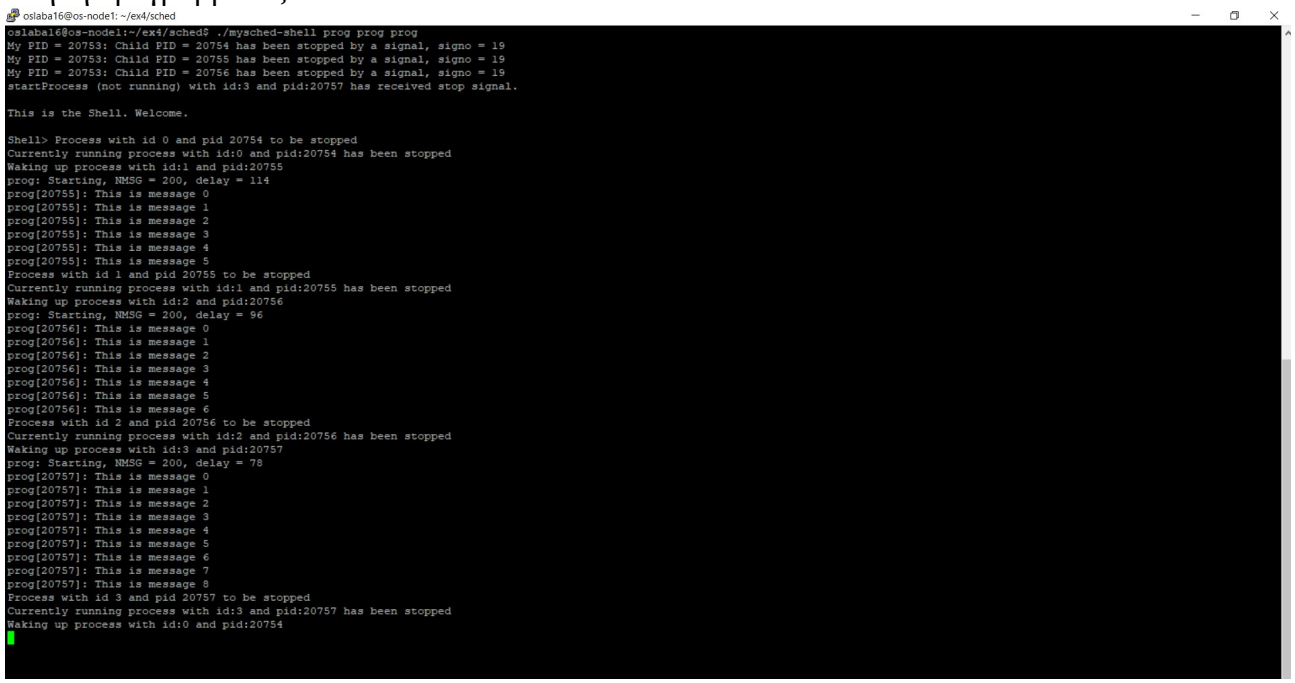
3.2 Άσκηση 1.2

Στην παρούσα άσκηση, υλοποιήσαμε ξανά έναν ασύγχρονο χρονοδρομολογητή κυκλικής επαναφοράς (RR) , ο οποίες χρησιμοποιεί σήματα για τον έλεγχο των διεργασιών, με ίδια λειτουργία με την προηγούμενη άσκηση, με την διαφορά όμως ότι στον παρόντα χρονοδρομολογητή προσθέσαμε την δυνατότητα αλληλεπίδρασης με τον φλοιό. Συγκεκριμένα, στις διεργασίες που ελέγχει ο χρονοδρομολογητής προστέθηκε μία νέα διεργασία, ονομαζόμενη shell (φλοιός), που αποτελεί την διεργασία που επιτρέπει στον χρήστη να δίνει ένα σύνολο εντολών κατά την διάρκεια της λειτουργίας του χρονοδρομολογητή. Οι εντολές αυτές λαμβάνονται από το πρόγραμμα του φλοιού και έτσι ο χρήστης μπορεί να ζητήσει από αυτόν την απαρίθμηση των διεργασιών, την δημιουργία μιας νέας διεργασίας, τον τερματισμό μιας διεργασίας ή τον τερματισμό της διεργασίας φλοιού. Για την υλοποίηση των τριών πρώτων αιτημάτων, υπεύθυνη δεν είναι όμως η διεργασία φλοιός, αλλά ο χρονοδρομολογητής, ο οποίος έχει τον έλεγχο όλων των διεργασιών, συμπεριλαμβανομένου του φλοιού. Επομένως, ο φλοιός μεταβιβάζει το αίτημα στον χρονοδρομολογητή, ο οποίος το διαχειρίζεται κατάλληλα. Τέλος, σημειώνεται ότι ο φλοιός χρονοδρομολογείται και αυτός με βάση τον κυκλικό αλγόριθμο του χρονοδρομολογητή, μαζί με τις υπόλοιπες διεργασίες και διαχειρίζεται τα αιτήματα του χρήστη στα χρονικά διαστήματα στα οποία παραχωρείται σε αυτός η CPU. Ο φλοιός είναι η πρώτη διεργασία που δημιουργούμε κατά την εκτέλεση του χρονοδρομολογητή και λαμβάνει το id 0.

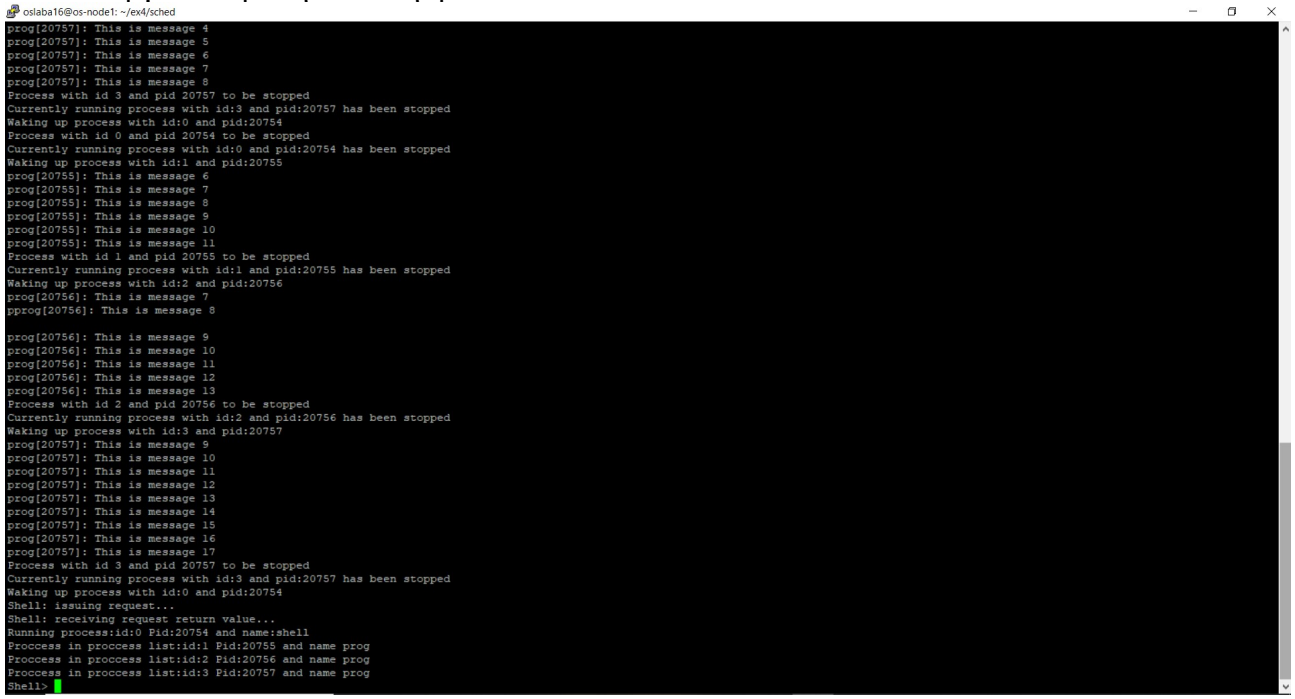
Ο κώδικας της άσκησης παρατίθεται στο τέλος της αναφοράς και στο αντίστοιχο αρχείο.

Ενδεικτικά παρατίθενται τμήματα της εξόδου του προγράμματος:

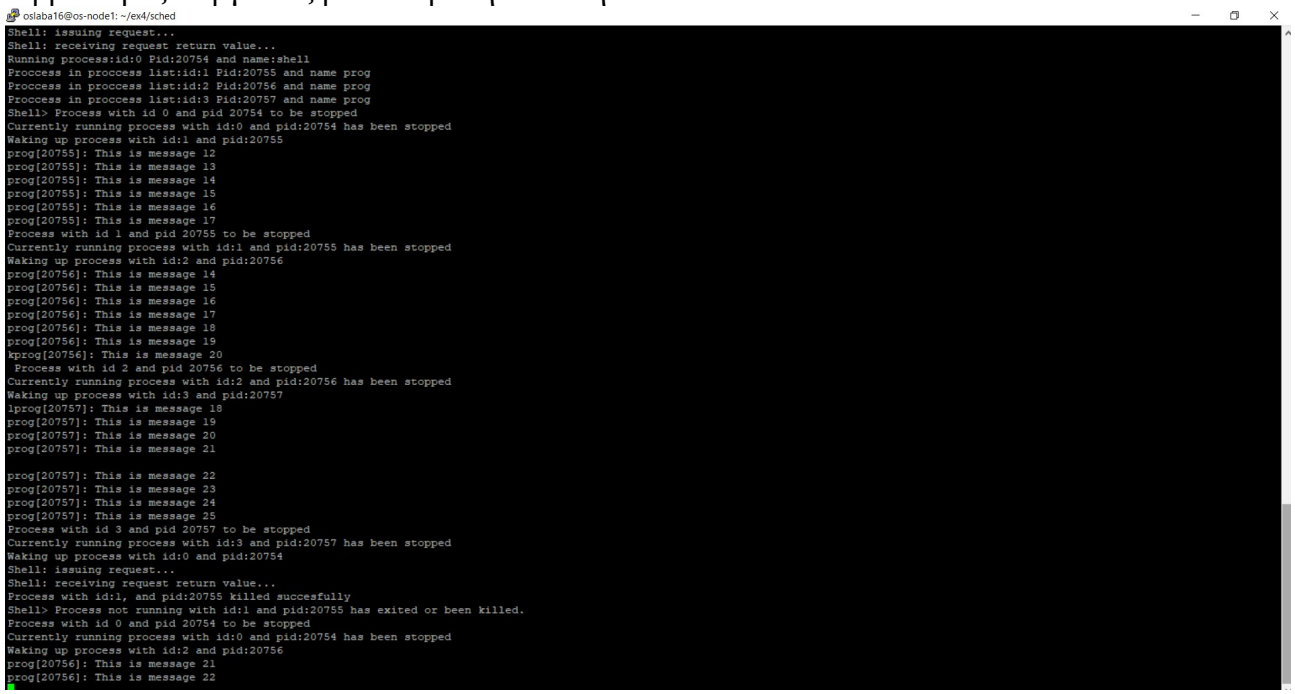
Κλήση προγράμματος:



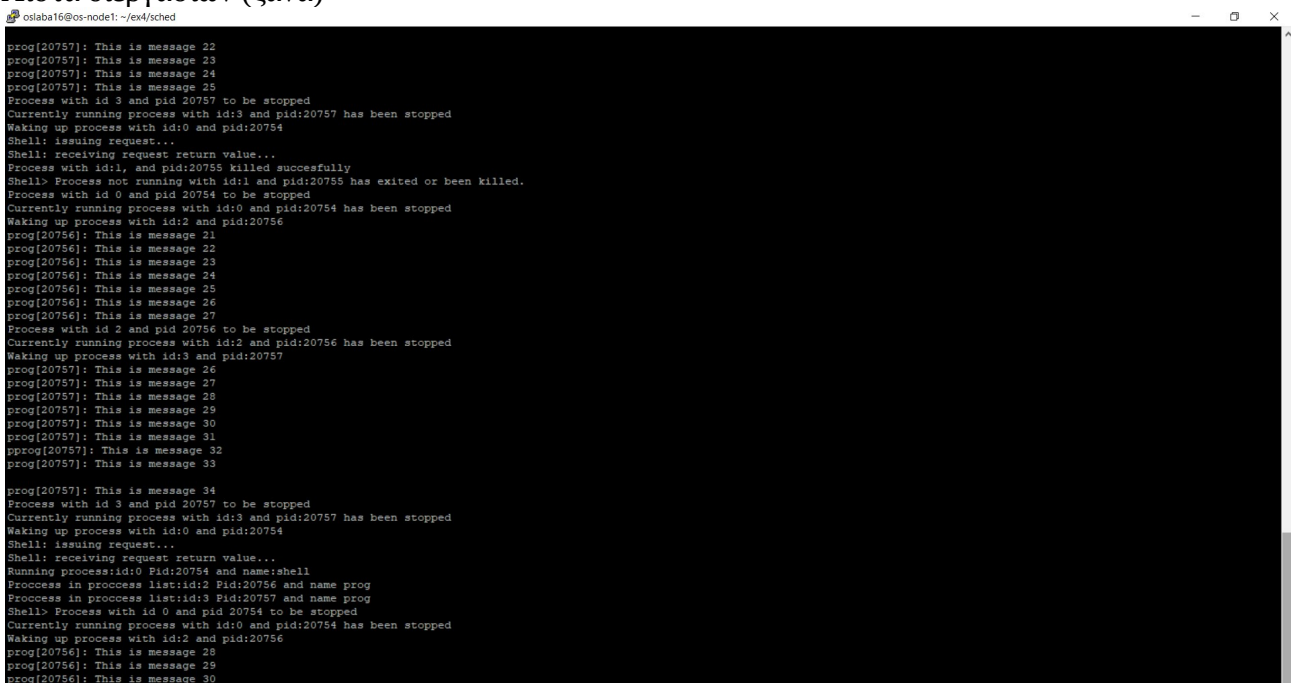
Λίστα διεργασιών με την εντολή: p



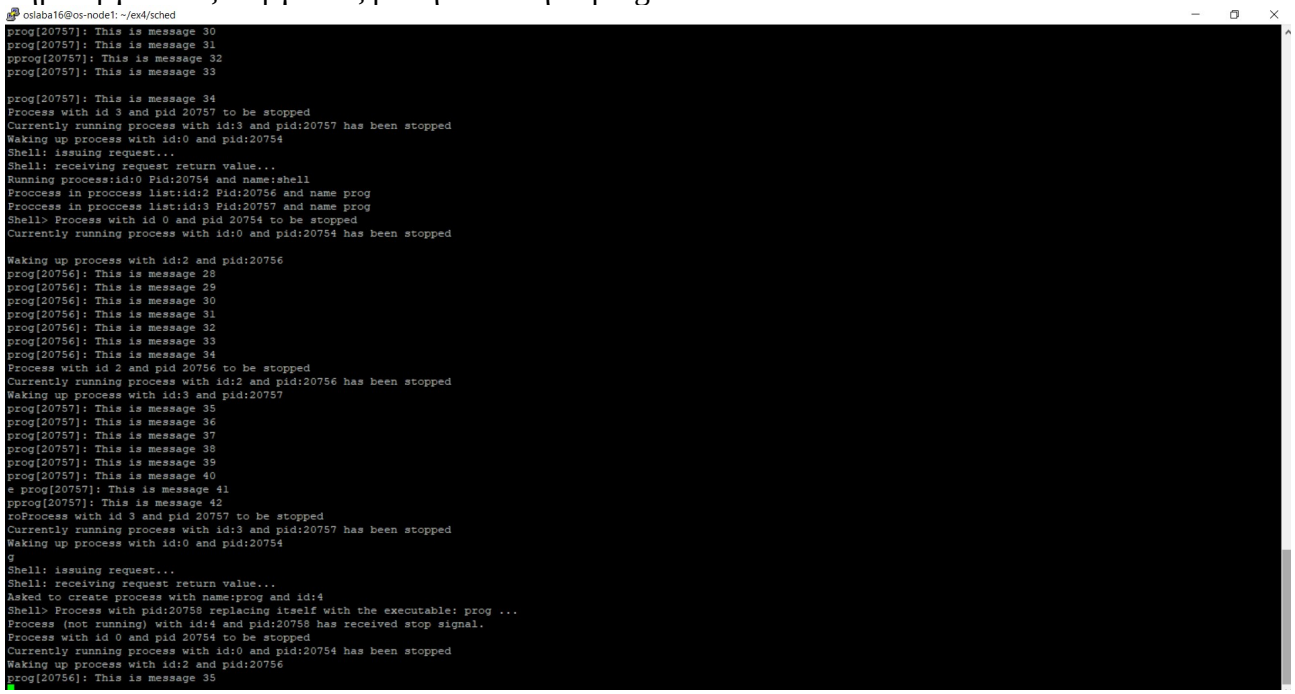
Τερματισμός διεργασίας με id=1 με την εντολή: k 1



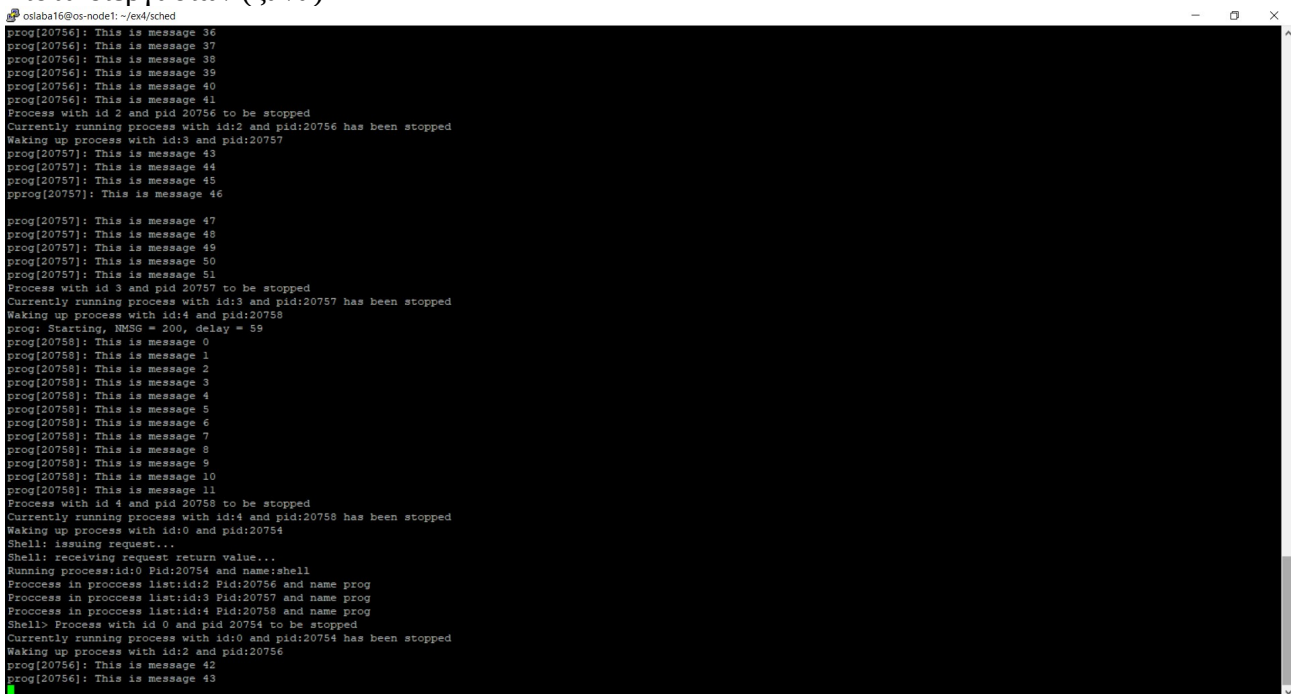
Λίστα διεργασιών (ξανά)



Δημιουργία νέας διεργασίας με την εντολή: e prog



Λίστα διεργασιών (ξανά)



Ερωτήσεις

1. Όταν ο φλοιός υφίσταται χρονοδρομολόγηση, όπως συμβαίνει στην περίπτωση μας και αναφέρθηκε ακριβώς πάνω, στην περίπτωση αιτήματος του χρήστη να δει την λίστα των διεργασιών με την εντολή ‘p’ στον φλοιό, ως τρέχουσα διεργασία στην λίστα εμφανίζεται πάντα ο φλοιός, κι αυτό διότι ο χρονοδρομολογητής υλοποιεί το αίτημα για απαρίθμηση των διεργασιών, όταν το λάβει από τον φλοιό, όταν ο φλοιός γίνει δηλαδή η τρέχουσα διεργασία, επεξεργαστεί το αίτημα του χρήση και το μεταβιβάσει στον χρονοδρομολογητή. Αυτό θα μπορούσε να μη συμβαίνει στην περίπτωση χρονοδρομολογούμενου φλοιού, μόνο αν το αίτημα για τύπωμα των διεργασιών δεν υλοποιούταν από τον χρονοδρομολογητή την στιγμή που το λαμβάνει, αλλά κάποια άλλη στιγμή, την οποία κρίνει αυτός κατάλληλη με βάση κάποια κριτήρια προτεραιοτήτων. Σε αυτή την

περίπτωση, θα μπορούσε όταν τελικά τυπωθούν οι διεργασίες να έχει εκπνεύσει το κβάντο χρόνου που αντιστοιχεί στον φλοιό και να έχει γίνει τρέχουσα κάποια άλλη διεργασία.

2. Όταν υλοποιούνται από τον δρομολογητή οι αιτήσεις του φλοιού είναι απαραίτητο να συμπεριλάβουμε τις κλήσεις των signals_disable(), _enable(), ώστε να μην είναι εφικτό κατά την διάρκεια υλοποίησης της αίτησης του φλοιού από τον χρονοδρομολογητή αυτός να λάβει την κάποιο από τα σήματα SIGCHILD και SIGALARM. Αυτό είναι απαραίτητο, γιατί όταν ζητάται στον χρονοδρομολογητή από τον φλοιό να δημιουργήσει μια νέα διεργασία ή να σκοτώσει μια διεργασία, τότε αυτός θα πρέπει στην πρώτη περίπτωση να δημιουργήσει μια διεργασία και να την προσθέσει στην λίστα διεργασιών και στην δεύτερη περίπτωση να αφαιρέσει την διεργασία από την λίστα διεργασιών και να της στείλει το σήμα SIGKILL . Αν ήταν εφικτό κατά την διάρκεια των παραπάνω ενεργειών να λάβει σήματα SIGCHILD ή SIGALARM, τότε θα μπορούσαν να προκύψουν σενάρια στα οποία ξεκίνησε ο handler χωρίς να έχει ολοκληρωθεί η υλοποίηση της αίτησης του φλοιού με αποτέλεσμα προβληματικές περιπτώσεις που οφείλονται στην διακοπή της εξυπηρέτησης της αίτησης του φλοιού, όπως το ενδεχόμενο μια διεργασία να δημιουργηθεί χωρίς ποτέ να εισαχθεί στην λίστα διεργασιών και άρα να εκτελεστεί .

3.3 Άσκηση 1.3

Στην άσκηση αυτή, προστίθενται στον χρονοδρομολογητή του προηγούμενου ερωτήματος δύο κλάσεις προτεραιότητας, η υψηλή και η χαμηλή προτεραιότητα. Αυτό επιτυγχάνεται στην υλοποίηση μας με την προσθήκη ενός στοιχείου που αφορά την κλάση προτεραιότητας στο pcb (process control block) κάθε διεργασίας. Όταν υπάρχουν διεργασίες με υψηλές προτεραιότητες, ο χρονοδρομολογητής δίνει κβάντα χρόνου με κυκλική χρονοδρομολόγηση μόνο σε αυτές, ενώ όταν υπάρχουν μόνο διεργασίες με χαμηλές προτεραιότητες, η χρονοδρομολόγηση γίνεται κανονικά όπως στα προηγούμενα ερωτήματα. Επίσης, όπως ζητάται, η προτεραιότητα όλων των διεργασιών τίθεται χαμηλή κατά την δημιουργία τους. Για την δυνατότητα αλλαγής των προτεραιοτήτων των διεργασιών, σύμφωνα με τα αιτήματα στον shell προσθέτουμε τις κατάλληλες περιπτώσεις (cases) στην συνάρτηση process_request, καθώς και τις κατάλληλες συναρτήσεις που θέτουν την προτεραιότητα μιας διεργασίας σε υψηλή ή χαμηλή και καλούνται από την process_request. (Η συνάρτηση που θέτει την προτεραιότητα μιας διεργασίας σε χαμηλή μπορεί να χρειαστεί να κληθεί και στον signal handler για το σήμα SIGCHLD σε περίπτωση που η διεργασία που τερμάτισε είχε υψηλή προτεραιότητα και δεν υπάρχουν άλλες διεργασίες εκτός του φλοιού με υψηλή προτεραιότητα ώστε η προτεραιότητα του φλοιού να γίνει χαμηλή. Αυτό θα γίνει κατανοητό στην επόμενη παράγραφο, όπου εξηγούμε πώς χειριζόμαστε την προτεραιότητα του φλοιού).

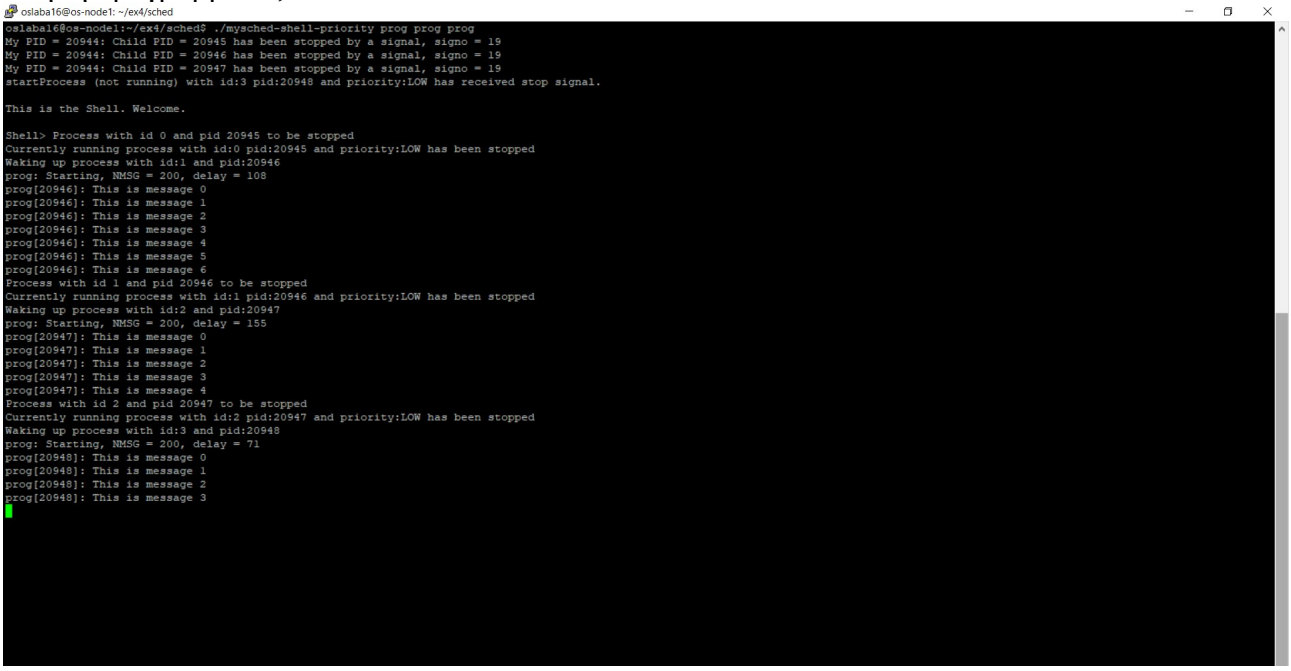
Ειδική περίπτωση αποτελεί ο φλοιός, καθώς θέλουμε να χρονοδρομολογείται μαζί με τις υπόλοιπες διεργασίες τόσο στην περίπτωση που υπάρχουν διεργασίες υψηλής προτεραιότητας, όσο και στην περίπτωση που υπάρχουν μόνο διεργασίες χαμηλής προτεραιότητας. Λύνουμε το πρόβλημα αυτό θέτοντας την προτεραιότητά του φλοιού ως χαμηλή κατά την δημιουργία του, όπως ορίζεται από την εκφώνηση (ο φλοιός είναι και αυτός διεργασία), αλλά τροποποιώντας την προτεραιότητά του ως εξής: Στην περίπτωση που υπάρχει τουλάχιστον μία διεργασία με υψηλή προτεραιότητα, ο φλοιός λαμβάνει και αυτός υψηλή προτεραιότητα, ώστε να συνεχίσει να τίθεται κυκλικά ως τρέχχουσα διεργασία. (Σε διαφορετική περίπτωση θα χάναμε την δυνατότητα να δίνουμε εντολές μόλις θέταμε μία διεργασία σε HIGH, αφού ο shell δεν θα γινόταν ποτέ ξανά running). Αν πάψουν αργότερα να υπάρχουν διεργασίες υψηλής προτεραιότητας, η προτεραιότητα του φλοιού δεν χρειάζεται πια να είναι HIGH και τίθεται ξανά σε LOW. Για να μην χάσουμε τον έλεγχο του προγράμματος από λάθος (χάνοντας την δυνατότητα να δίνουμε εντολές στον φλοιό και άρα να δημιουργούμε , να τερματίζουμε διεργασίες κ.α.) όταν υπάρχουν διεργασίες υψηλής προτεραιότητας δεν επιτρέπουμε στον χρήστη να θέσει την προτεραιότητα του φλοιού σε χαμηλή. Τέλος, στην περίπτωση που έχουμε μόνο διεργασίες χαμηλής προτεραιότητας και η προτεραιότητα

του φλοιού τεθεί υψηλή, εκτελείται μόνο αυτός, μέχρι να θέσουμε ξανά την προτεραιότητά του ως χαμηλή, ή να θέσουμε την προτεραιότητα κάποιας άλλης διεργασίας ως υψηλή.

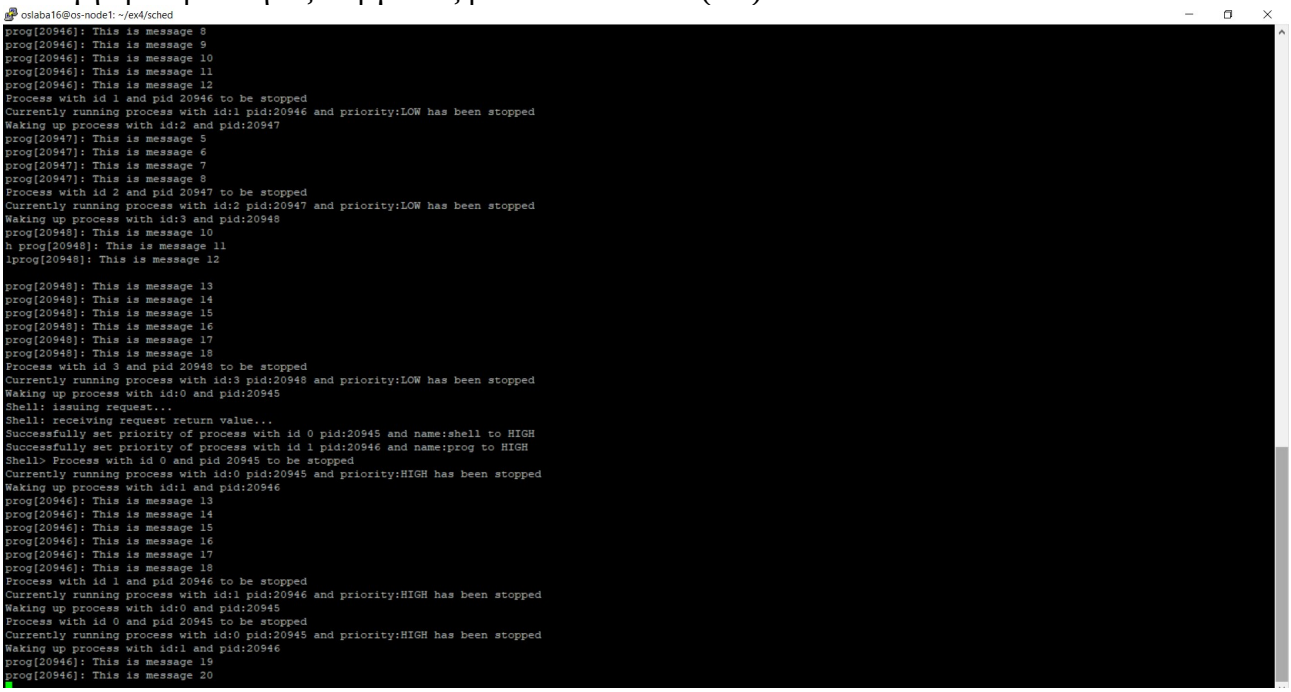
Ο κώδικας της άσκησης παρατίθεται στο τέλος της αναφοράς και στο αντίστοιχο αρχείο.

Στη συνέχεια παρατίθενται στιγμιότυπα της εξόδου του προγράμματος ενδεικτικά:

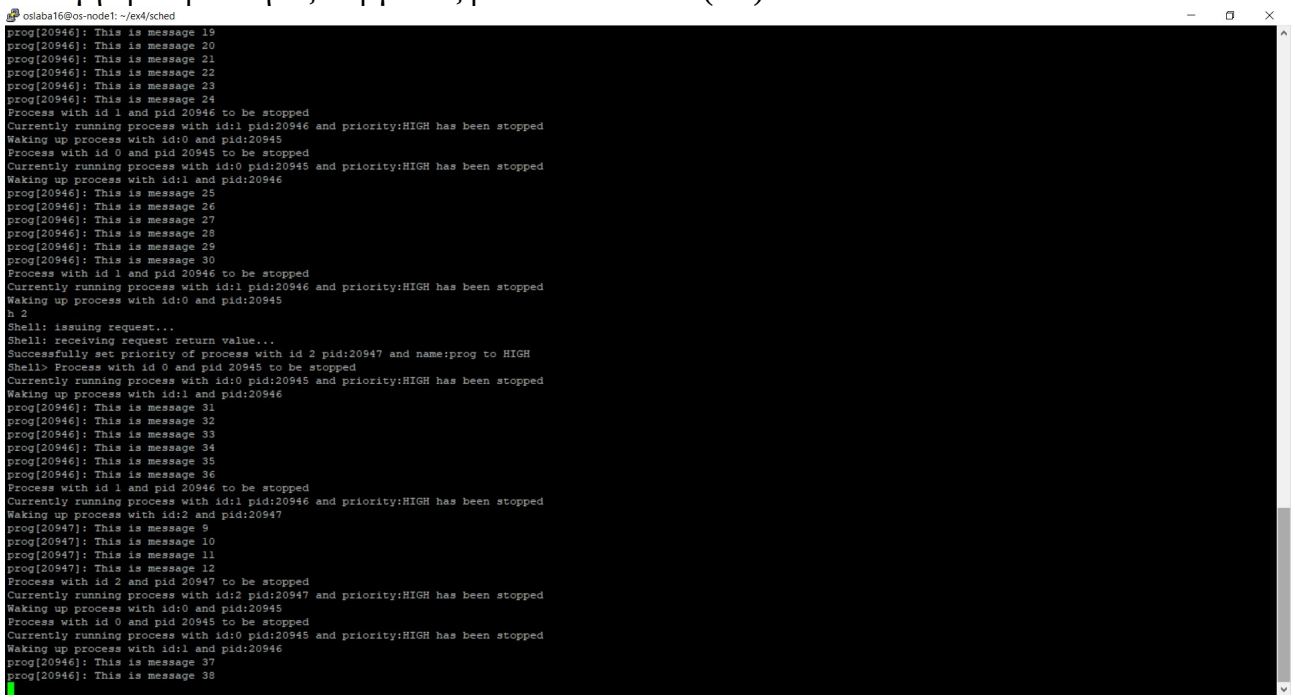
Κλήση προγράμματος:



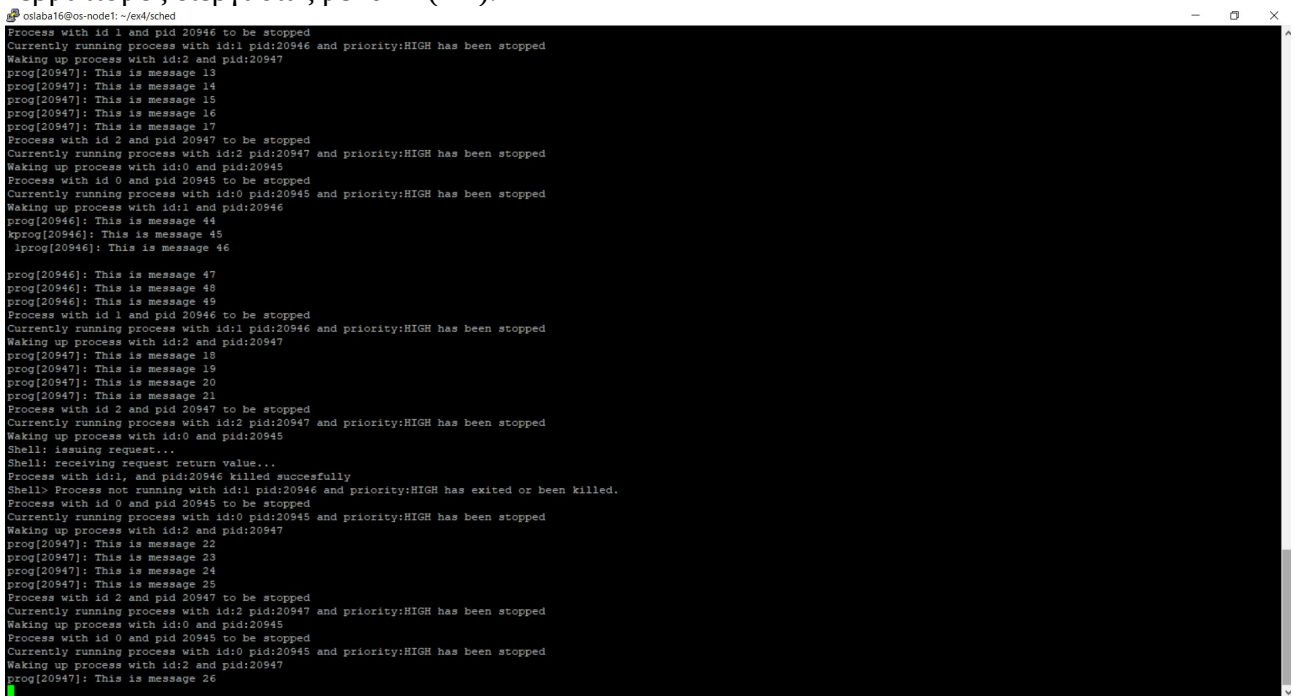
Αλλαγή προτεραιότητας διεργασίας με id=1 σε HIGH (h 1).



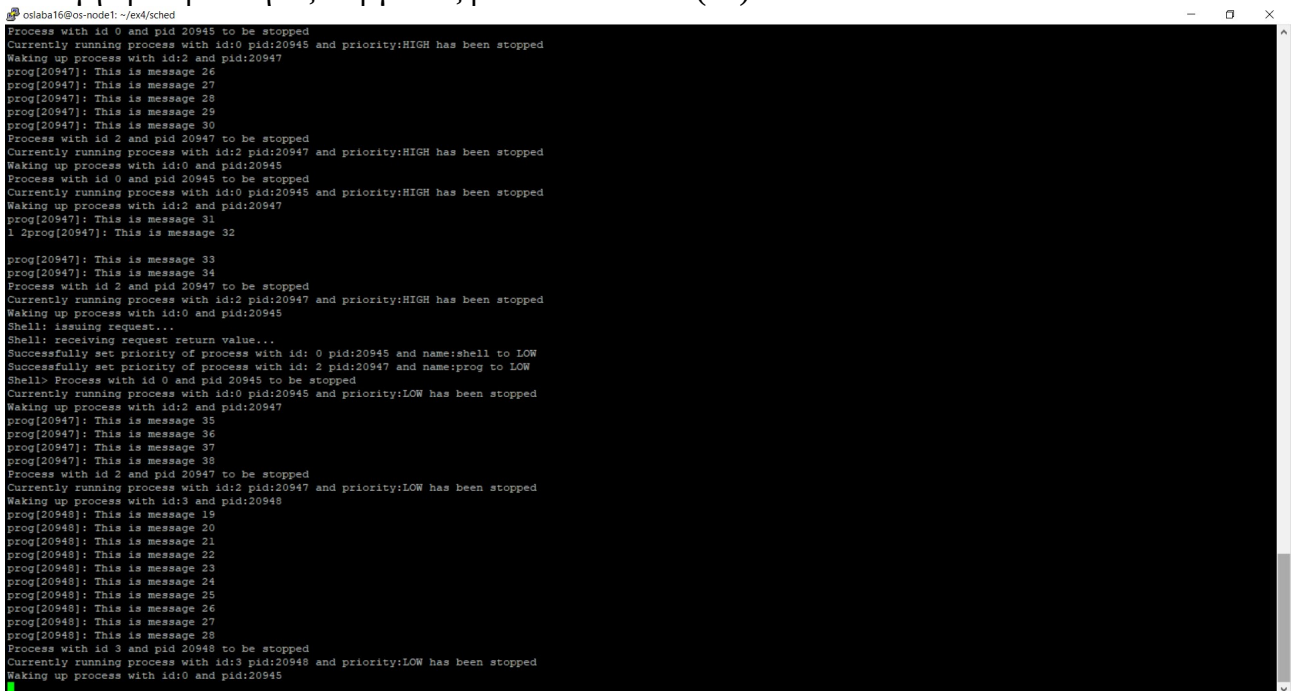
Αλλαγή προτεραιότητας διεργασίας με id=2 σε HIGH (h 2).



Τερματισμός διεργασίας με id=1 (k 1).



Αλλαγή προτεραιότητας διεργασίας με id=2 σε LOW (l 2).



Ερωτήσεις:

1. Ένα σενάριο λιμοκτονίας είναι η περίπτωση που μία διεργασία δεν λαμβάνει ποτέ υψηλή προτεραιότητα, ενώ υπάρχουν πάντα διεργασίες με υψηλή προτεραιότητα, με αποτέλεσμα η διεργασία με χαμηλή προτεραιότητα να μην γίνεται τρέχουσα ποτέ. Ένα συγκεκριμένο παράδειγμα είναι το εξής: Έστω ότι εκτός του δρομολογητή και του φλοιού δημιουργούνται οι διεργασίες με ids 1 2 3 και 4. Μετά την δημιουργία τους, τίθεται η προτεραιότητα των 1 2 ως υψηλή με αποτέλεσμα να χρονοδρομολογούνται διαρκώς οι 1 2 και ο shell. Έτσι, οι 3 και 4 που έχουν από την δημιουργία τους χαμηλή προτεραιότητα δεν εκτελείται. Στην συνέχεια, η διεργασία 4 ολοκληρώνεται ή λαμβάνει κάποιο σήμα kill, αλλά δημιουργείται μια νέα διεργασία με id 6 (αν θεωρήσουμε ότι το id 5 ανήκει στον φλοιό και το id 0 στον χρονοδρομολογητή) και η προτεραιότητά της τίθεται υψηλή. Αργότερα πεθαίνει και αυτή αλλά αντικαθίσταται από μια ή περισσότερες νέες με υψηλή προτεραιότητα. Το μοτίβο συνεχίζει έτσι και η διεργασία 3, αν δεν λάβει ποτέ υψηλή προτεραιότητα δεν θα εκτελεστεί ποτέ.

Στην συνέχεια παρατίθεται ο κώδικας των τριών ασκήσεων:

Άσκηση 1.1 (αρχείο “myscheduler.c”):

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

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

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

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

typedef struct proc_struct {  //pcb struct
        pid_t pid;
        int id;
        char process_name[TASK_NAME_SZ];
        }process;

static int number_of_processes;

typedef struct proc_list{    //list node struct
        process *proc_pointer;
        struct proc_list *next;
        }proc_node;

static proc_node *running_process;  //runing process is kept here, processes are a circular list


static void
sigalrm_handler(int signum)
{
        printf("Process with id %d and pid %d to be stopped \n", (running_process->proc_pointer)-
>id, (running_process->proc_pointer)->pid);
        if (kill((running_process->proc_pointer)->pid, SIGSTOP) <0) {
                perror("scheduler kill error");
                exit(1);
        }

//      assert(0 && "Please fill me!");
}

proc_node* get_process_with_pid( pid_t pid) {
        proc_node *pr=running_process;
        while ((pr->proc_pointer)->pid!=pid) {
                pr=pr->next;
                if (pr==running_process) {
                        return NULL;
                }
        }
        return pr;

}
/*
* SIGCHILD handler
*/
static void
sigchld_handler(int signum)
{
        pid_t p;
```

```
	int run_next;
	int status;
	for (;;) {
		run_next=0;
		p=waitpid(-1, &status, WUNTRACED | WNOHANG );
		if (p<0) {
			perror("Waitpid<0 in sigchld_handler");
			exit(1);
		}
		if (p==0)
			break;
		if (WIFSTOPPED(status)) {
			if (p==(running_process->proc_pointer)->pid){
				printf("Currently running process with id:%d and pid:%d has been
stopped\n",(running_process->proc_pointer)->id, (running_process->proc_pointer)->pid);
				run_next=1;
				running_process=running_process->next;
			}
			else {  //should not go here: only the running task is stopped, by the scheduler
				proc_node *pr=get_process_with_pid(p);
				if (pr==NULL)
					printf("Error getting data of stopped process.");
				else {
					printf("Process (not running) with id:%d and pid:%d has
received stop signal.\n", (pr->proc_pointer)->id, (pr->proc_pointer)->pid);
				}
			}
		}
		if (WIFEXITED(status) || WIFSIGNALED(status)) {
			if (p==(running_process->proc_pointer)->pid) {
				printf("Currently running process with id:%d and pid:%d has exited or
been killed.\n", (running_process->proc_pointer)->id, (running_process->proc_pointer)->pid);
				run_next=1;
			if (running_process->next==running_process) {  //killed process was the last
one, no more child processes
				printf("Scheduler: No more more processes to run.\n");
				exit(0);
			}
			}
			else {
				proc_node *pr=get_process_with_pid(p);
				printf("Process not running with id:%d and pid:%d has exited or been
killed.\n",(pr->proc_pointer)->id, (pr->proc_pointer)->pid);
			if (pr==pr->next) {  //killed process was the last one, no more child processes
				printf("Scheduler: No more processes to run.\n");
				exit(0);
			}
			}
			proc_node *rem=running_process;
			while (((rem->next)->proc_pointer)->pid!=p)  //remove killed process from
process list
				rem=rem->next;
```

```
                        proc_node *tmp=rem->next;
                        rem->next=(rem->next)->next;
                        free(tmp->proc_pointer);
                        free(tmp);
                        if (run_next==1)
                                running_process=running_process->next;
                }
                if (run_next==1) {   //if killed process is the current process, find the next one
                        printf("Waking up process with id:%d and pid:%d\n", (running_process-
>proc_pointer)->id, (running_process->proc_pointer)->pid);
                        kill((running_process->proc_pointer)->pid,SIGCONT);
                        alarm(SCHED_TQ_SEC);
                }
        }
        //assert(0 && "Please fill me!");
}

/* 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);
```

```
	}
}

int main(int argc, char *argv[])
{
	printf("start");
	int nproc;
	nproc=argc-1;
	number_of_processes=nproc; //input process (shell will receive pid 0)

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

	//nproc = 0; /* number of proccesses goes here */

	int i;
	for (i=1; i<=nproc; i++) {
		pid_t pid;
		pid=fork();
		char *argvs[] = { argv[i], NULL};
		char *newenviron[]={NULL};
		if (pid<0) {
			perror("main: fork");
			exit(1);
		}
		if (pid==0) {
			raise(SIGSTOP);
			execve( argv[i], argvs, newenviron);
			perror("main: execve");
			exit(1);
		}
		if (pid>0) {
			proc_node *pn=(proc_node*)malloc(sizeof(proc_node));
			process *pr=(process*)malloc(sizeof(process));
			pr->id=i;
			pr->pid=pid;
			strcpy(pr->process_name, argv[i]);
			pn->proc_pointer=pr;
			if (i==1) {
			running_process=pn; //initialize pointer of running_process which is out
pointer to the list of processed
			running_process->next=running_process;
			}
			pn->next=running_process->next;
			running_process->next=pn;
			running_process=pn;


		}
	}
```

```
	running_process=running_process->next; //running process pointer points to the last process we added, make running process the first process that was added


	/* 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) {  //we don't use this part later, exit is already handled in SIGCHLD signal handler, this part only used if no task arguements are provided
		fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
		exit(1);
	}
	if (nproc>0) {
		kill((running_process->proc_pointer)->pid,SIGCONT);
		alarm(SCHED_TQ_SEC);
	}


	while (pause())
		;

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

Άσκηση 1.2 (αρχείο “mysched-shell.c”):

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

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

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

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

typedef struct proc_struct { //pcb struct
        pid_t pid;
        int id;
        char process_name[TASK_NAME_SZ];
        }process;

static int number_of_processes;

typedef struct proc_list{  //list node struct
        process *proc_pointer;
        struct proc_list *next;
        }proc_node;

static proc_node *running_process;  //runing process is kept here, processes are a circular list

/* Print a list of all tasks currently being scheduled.  */
static void
sched_print_tasks(void)
{
        printf("Running process:id:%d Pid:%d and name:%s \n",(running_process->proc_pointer)-
>id, (running_process->proc_pointer)->pid, (running_process->proc_pointer)->process_name);
        proc_node *p=running_process;
        p=p->next;
        while (p!=running_process) {
        printf("Proccess in proccess list:id:%d Pid:%d and name %s \n",(p->proc_pointer)->id, (p-
>proc_pointer)->pid, (p->proc_pointer)->process_name);
        p=p->next;
        }
        //assert(0 && "Please fill me!");
}
```

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


static int
sched_kill_task_by_id(int id)
{
	proc_node *p=running_process->next;
	while (p!=running_process && (p->proc_pointer)->id!=id)  //case of shell:p=running process, other cases: loop until the asked id is found
		p=p->next;
	if (p==running_process && id!=0) {  //case of shell: p==running process, id==0: don't enter, other cases: scanned all the list, not found: return
		printf("Coud not find process that was asked to kill, exiting...\n");
		printf("asked id:%d\n", id);
		return 1;
	}
	if ((p->proc_pointer)->id==id){
		kill((p->proc_pointer)->pid,SIGKILL);
		printf("Process with id:%d, and pid:%d killed succesfully\n", (p->proc_pointer)->id, (p->proc_pointer)->pid);
		return 0;
	}

	//assert(0 && "Please fill me!");
	//return -ENOSYS;
return 0;
}


/* Create a new task.  */
static void
sched_create_task(char *executable)
{
	printf("Asked to create process with name:%s and id:%d\n", executable, number_of_processes+1);
	pid_t pid=fork();
	if (pid<0){
		perror("sched_create_task fork error");
	}
	if (pid==0) {  //child replaces itself
		char *newargv[]={executable,NULL,NULL,NULL};
		char *newenviron[]={NULL};
		printf("Process with pid:%ld replacing itself with the executable: %s ...\n", (long)getpid(), executable);
		raise(SIGSTOP);
		execve(executable, newargv, newenviron);
		perror("shed_create_task execve error");
		exit(1);
	}
	if (pid>0){  //parent keeps count of processes, adds child to the processes' list
```

```
                number_of_processes++;
                proc_node *pn=( proc_node *) malloc(sizeof(proc_node));
                process *p=(process *) malloc(sizeof(process));
                p->id=number_of_processes;
                p->pid=pid;
                strcpy(p->process_name, executable);
                proc_node *tmp=running_process;
                while (tmp->next!=running_process) {  //add process to the end of the list, so before
the running process
                        tmp=tmp->next;
                }
                tmp->next=pn;
                pn->next=running_process;
                pn->proc_pointer=p;
        }


//      assert(0 && "Please fill me!");
}

/* 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)
{
        printf("Process with id %d and pid %d to be stopped \n", (running_process->proc_pointer)-
>id, (running_process->proc_pointer)->pid);
        if (kill((running_process->proc_pointer)->pid, SIGSTOP) <0) {
                perror("scheduler kill error");
                exit(1);
```

```
	}

//	assert(0 && "Please fill me!");
}

proc_node* get_process_with_pid( pid_t pid) {
	proc_node *pr=running_process;
	while ((pr->proc_pointer)->pid!=pid) {
		pr=pr->next;
		if (pr==running_process) {
			return NULL;
		}
	}
	return pr;

}
/*
* SIGCHILD handler
*/
static void
sigchld_handler(int signum)
{
	pid_t p;
	int run_next;
	int status;
	for (;;) {
		run_next=0;
		p=waitpid(-1, &status, WUNTRACED | WNOHANG );
		if (p<0) {
			perror("Waitpid<0 in sigchld_handler");
			exit(1);
		}
		if (p==0)
			break;
		if (WIFSTOPPED(status)) {
			if (p==(running_process->proc_pointer)->pid){
				printf("Currently running process with id:%d and pid:%d has been
stopped\n",(running_process->proc_pointer)->id, (running_process->proc_pointer)->pid);
				run_next=1;
				running_process=running_process->next;
			}
			else {	//should not go here: only the running task is stopped, by the scheduler
				proc_node *pr=get_process_with_pid(p);
				if (pr==NULL)
					printf("Error getting data of stopped process.");
				else {
					printf("Process (not running) with id:%d and pid:%d has
received stop signal.\n", (pr->proc_pointer)->id, (pr->proc_pointer)->pid);
				}
			}
		}
		if (WIFEXITED(status) || WIFSIGNALED(status)) {
```

```
                        if (p==(running_process->proc_pointer)->pid) {
                                printf("Currently running process with id:%d and pid:%d has exited or
been killed.\n", (running_process->proc_pointer)->id, (running_process->proc_pointer)->pid);
                                run_next=1;
                                if (running_process==running_process->next) {
                                        printf("Scheduler:No more processes to run.\n");
                                        exit(0);
                                }
                        }
                        else {
                                proc_node *pr=get_process_with_pid(p);
                                printf("Process not running with id:%d and pid:%d has exited or been
killed.\n",(pr->proc_pointer)->id, (pr->proc_pointer)->pid);
                                if (pr==pr->next) { //process that exited or was killed was the last
process
                                        printf("Scheduler: No more processes to run.\n");
                                        exit(0);
                                }
                        }
                        proc_node *rem=running_process;
                        while (((rem->next)->proc_pointer)->pid!=p)  //remove killed process from
process list
                                rem=rem->next;
                        proc_node *tmp=rem->next;
                        rem->next=(rem->next)->next;
                        free(tmp->proc_pointer);
                        free(tmp);
                        if (run_next==1)
                                running_process=running_process->next;
                }
                if (run_next==1) {   //if killed process is the current process, find the next one
                        printf("Waking up process with id:%d and pid:%d\n", (running_process-
>proc_pointer)->id, (running_process->proc_pointer)->pid);
                        kill((running_process->proc_pointer)->pid,SIGCONT);
                        alarm(SCHED_TQ_SEC);
                }
        }
        //assert(0 && "Please fill me!");
}

/* 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);
        }
}

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

        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);
}

/* 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 void
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 */

        running_process=(proc_node *) malloc(sizeof(proc_node));  //shell is added to the process
list as the running process
        process *proc=(process *) malloc(sizeof(process));
        running_process->proc_pointer=proc;
        proc->id=0;
        proc->pid=p;
        running_process->next=running_process;
        strcpy(proc->process_name,SHELL_EXECUTABLE_NAME);
        close(pfds_rq[1]);
        close(pfds_ret[0]);
        *request_fd = pfds_rq[0];
        *return_fd = pfds_ret[1];
}

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[])
{
        printf("start");
        int nproc;
        /* Two file descriptors for communication with the shell */
```

```
	static int request_fd, return_fd;
	nproc=argc-1;
	number_of_processes=nproc; //input process (shell will receive pid 0)
	/* Create the shell. */
	sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd, &return_fd);
	/* TODO: add the shell to the scheduler's tasks */

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

	//nproc = 0; /* number of proccesses goes here */

	int i;
	for (i=1; i<=nproc; i++) {
		pid_t pid;
		pid=fork();
		char *argvs[] = { argv[i], NULL};
		char *newenviron[]={NULL};
		if (pid<0) {
			perror("main: fork");
			exit(1);
		}
		if (pid==0) {
			raise(SIGSTOP);
			execve( argv[i], argvs, newenviron);
			perror("main: execve");
			exit(1);
		}
		if (pid>0) {
			proc_node *pn=(proc_node*)malloc(sizeof(proc_node));
			process *pr=(process*)malloc(sizeof(process));
			pr->id=i;
			pr->pid=pid;
			strcpy(pr->process_name, argv[i]);
			pn->next=running_process->next;
			pn->proc_pointer=pr;
			running_process->next=pn;
			running_process=pn;


		}
	}
	running_process=running_process->next; //running_process was the last processes we
addeed, make running process the first one (FIFO)


	/* 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) {					//We don't need this part anymore!!! There is always a process when we call the program, the shell.
//		fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
//		exit(1);
//	}
//	if (nproc>0) {
		kill((running_process->proc_pointer)->pid,SIGCONT);   //always wake a process up, we always have at least the shell
		alarm(SCHED_TQ_SEC);
//	}


	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;
}
```

Άσκηση 1.3 (αρχείο “mysched-shell-priority.c”):

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

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

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

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

typedef struct proc_struct {  //pcb struct
        pid_t pid;
        int id;
        char process_name[TASK_NAME_SZ];
        char priority[5];
        }process;

static int number_of_processes;  //list node struct

typedef struct proc_list{
        process *proc_pointer;
        struct proc_list *next;
        }proc_node;

static proc_node *running_process;  //running process is kept here, list of procesess is a circular list
static int num_high=0;

/* Print a list of all tasks currently being scheduled.  */
static void
sched_print_tasks(void)
{
        printf("Running process:id:%d Pid:%d name:%s priority:%s \n",(running_process-
>proc_pointer)->id, (running_process->proc_pointer)->pid, (running_process->proc_pointer)-
>process_name, (running_process->proc_pointer)->priority);
        proc_node *p=running_process;
        p=p->next;
        while (p!=running_process) {
        printf("Proccess in proccess list:id:%d Pid:%d name %s priority: %s\n",(p->proc_pointer)-
>id, (p->proc_pointer)->pid, (p->proc_pointer)->process_name, (p->proc_pointer)->priority);
        p=p->next;
        }
```

```
	//assert(0 && "Please fill me!");
}


/* Send SIGKILL to a task determined by the value of its
 * scheduler-specific id.
 */
static int
sched_kill_task_by_id(int id)
{
	proc_node *p=running_process->next;
	while (p!=running_process && (p->proc_pointer)->id!=id)
		p=p->next;
	if (p==running_process && id!=0) {
		printf("Coud not find process that was asked to kill, exiting...\n");
		printf("asked id:%d\n", id);
		return 1;
	}
	if ((p->proc_pointer)->id==id){
		kill((p->proc_pointer)->pid,SIGKILL);
		printf("Process with id:%d, and pid:%d killed succesfully\n", (p->proc_pointer)->id,
(p->proc_pointer)->pid);
		return 0;
	}

	//assert(0 && "Please fill me!");
	//return -ENOSYS;
return 0;
}


/* Create a new task.  */
static void
sched_create_task(char *executable)
{
	printf("Asked to create process with name:%s and id:%d\n", executable,
number_of_processes+1);
	pid_t pid=fork();
	if (pid<0){
		perror("sched_create_task fork error");
	}
	if (pid==0) {  //child replaces itself
		char *newargv[]={executable,NULL,NULL,NULL};
		char *newenviron[]={NULL};
		printf("Process with pid:%ld replacing itself with the executable: %s ...\n",
(long)getpid(), executable);
		raise(SIGSTOP);
		execve(executable, newargv, newenviron);
		perror("shed_create_task execve error");
		exit(1);
	}
	if (pid>0){  //parent keeps count of processes, adds child to the processes' list
```

```
                number_of_processes++;
                proc_node *pn=( proc_node *) malloc(sizeof(proc_node));
                process *p=(process *) malloc(sizeof(process));
                p->id=number_of_processes;
                p->pid=pid;
                strcpy(p->process_name, executable);
                proc_node *tmp=running_process;
                while (tmp->next!=running_process) {  //add process to the end of the list, so before
the running process
                        tmp=tmp->next;
                }
                tmp->next=pn;
                pn->next=running_process;
                pn->proc_pointer=p;
                strcpy(pn->proc_pointer->priority, "LOW");

        }


//      assert(0 && "Please fill me!");
}
static void sched_set_low_priority(int id) {
        proc_node *p=running_process;
        if (num_high>1 && id==0) {
                printf("Cannot change shells priority to LOW, because shell control will not be able
to be regained.\n");
                return;
        }
        while ((p->proc_pointer)->id!=id){
                p=p->next;
                if (p==running_process) {
                        printf("Error finding process to set priority to LOW\n");
                        return;
                }
        }
        if (strcmp(p->proc_pointer->priority, "HIGH")==0)  //if process had high priority before,
number of processes with high priority (shell excluded) is decreased;
                num_high--;
        if (num_high==1 && id!=0) //if there is only one high priority process it is the shell, shell
must have low priority (no nead to call again for shell)
                sched_set_low_priority(0);
        strcpy(p->proc_pointer->priority,"LOW");

        printf("Successfully set priority of process with id: %d pid:%d and name:%s to LOW\n", (p-
>proc_pointer)->id, (p->proc_pointer)->pid, (p->proc_pointer)->process_name);
}
static void sched_set_high_priority(int id){
        proc_node *p=running_process;
        while ((p->proc_pointer)->id!=id){
                p=p->next;
                if (p==running_process) {
                        printf("Error finding process to set priority to HIGH\n");
```

```
                return;
            }
        }
        if (num_high==0 && id!=0) //if it is the first process getting high priority, the shell must also get high priority (we don't need to call the function again if we are already changing shell's priority)
            sched_set_high_priority(0);
        if (strcmp(p->proc_pointer->priority, "LOW")==0)  //if process had low priority before, number of processes with high priority is increased
            num_high++;
        strcpy(p->proc_pointer->priority,"HIGH");
        printf("Successfully set priority of process with id %d pid:%d and name:%s to HIGH\n", (p->proc_pointer)->id, (p->proc_pointer)->pid, (p->proc_pointer)->process_name);
}

/* 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_LOW_TASK:
                sched_set_low_priority(rq->task_arg);
                return 0;
            case REQ_HIGH_TASK:
                sched_set_high_priority(rq->task_arg);
                return 0;
            default:
                return -ENOSYS;
        }
}

/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum)
{
        printf("Process with id %d and pid %d to be stopped \n", (running_process->proc_pointer)->id, (running_process->proc_pointer)->pid);
        if (kill((running_process->proc_pointer)->pid, SIGSTOP) <0) {
            perror("scheduler kill error");
```

```
                exit(1);
        }

//      assert(0 && "Please fill me!");
}

proc_node* get_process_with_pid( pid_t pid) {
        proc_node *pr=running_process;
        while ((pr->proc_pointer)->pid!=pid) {
                pr=pr->next;
                if (pr==running_process) {
                        return NULL;
                }
        }
        return pr;

}
/*
* SIGCHILD handler
*/
static void
sigchld_handler(int signum)
{
        pid_t p;
        int run_next;  //helper variables
        int status;

        for (;;) {
                run_next=0;
                p=waitpid(-1, &status, WUNTRACED | WNOHANG );
                if (p<0) {
                        perror("Waitpid<0 in sigchld_handler");
                        exit(1);
                }
                if (p==0)
                        break;
                if (WIFSTOPPED(status)) {
                        if (p==(running_process->proc_pointer)->pid){
                                printf("Currently running process with id:%d pid:%d and priority:%s
has been stopped\n",(running_process->proc_pointer)->id, (running_process->proc_pointer)->pid,
(running_process->proc_pointer)->priority);
                                run_next=1;
                                running_process=running_process->next;
                        }
                        else {  //should not go here: only the running task is stopped, by the scheduler
                                proc_node *pr=get_process_with_pid(p);
                                if (pr==NULL)
                                        printf("Error getting data of stopped process.");
                                else {
```

```
                                        printf("Process (not running) with id:%d pid:%d and priority:
%s has received stop signal.\n", (pr->proc_pointer)->id, (pr->proc_pointer)->pid, (pr-
>proc_pointer)->priority);
                                }
                        }
                }
                if (WIFEXITED(status) || WIFSIGNALED(status)) {

                        if (p==(running_process->proc_pointer)->pid) {
                                printf("Currently running process with id:%d pid:%d and priority:%s
has exited or been killed.\n", (running_process->proc_pointer)->id, (running_process-
>proc_pointer)->pid, running_process->proc_pointer->priority);
                                run_next=1;
                        if (strcmp(running_process->proc_pointer->priority,"HIGH")==0) {  //if
priority of process was high
                                num_high--;                                     //reduce the
number of processes with high priority
                                if (num_high==1 && running_process->proc_pointer->id!=0)
sched_set_low_priority(0);    //if only one process has high priority (the shell) set shell's priority to
low
                        }
                        if (running_process==running_process->next) {
                                printf("Scheduler: No more processes to run.\n");
                                exit(0);
                        }
                        }
                        else {
                                proc_node *pr=get_process_with_pid(p);
                                printf("Process not running with id:%d pid:%d and priority:%s has
exited or been killed.\n",(pr->proc_pointer)->id, (pr->proc_pointer)->pid, pr->proc_pointer-
>priority);
                                if (strcmp(pr->proc_pointer->priority,"HIGH")==0) {
                                        num_high--;
                                        if (num_high==1 && pr->proc_pointer->id!=0)
sched_set_low_priority(0);
                                }
                                if (pr==pr->next) {
                                        printf("Scheduler: No more processes to run.\n");
                                        exit(0);
                                }
                        }
                        proc_node *rem=running_process;
                        while (((rem->next)->proc_pointer)->pid!=p)  //remove killed process from
process list
                                rem=rem->next;
                        proc_node *tmp=rem->next;
                        rem->next=(rem->next)->next;
                        free(tmp->proc_pointer);
                        free(tmp);
                        if (run_next==1)
                                running_process=running_process->next;
                }
```

```
                if (run_next==1) {   //if killed process is the current process, find the next one
                        proc_node *pr=running_process;
                        //if ((pr->proc_pointer)->id!=0) {  //if id=0 it is the shell, we don't care about
its priority
                        if (num_high!=0) {  //if there are processes with high priority, choose the next
one with high prioriry
                        while ((strcmp(pr->proc_pointer->priority, "LOW")==0)){
                                        pr=pr->next;
                                }
                        }
                        running_process=pr;
                        printf("Waking up process with id:%d and pid:%d\n", (running_process-
>proc_pointer)->id, (running_process->proc_pointer)->pid);
                        kill((running_process->proc_pointer)->pid,SIGCONT);
                        alarm(SCHED_TQ_SEC);
                }
        }
        //assert(0 && "Please fill me!");
}

/* 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);
	}
}

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

	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);
}

/* 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 void
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 */

	running_process=(proc_node *) malloc(sizeof(proc_node));  //shell is added to the process
list as the running process
	process *proc=(process *) malloc(sizeof(process));
	running_process->proc_pointer=proc;
	proc->id=0;
	proc->pid=p;
	running_process->next=running_process;
	strcpy(proc->process_name,SHELL_EXECUTABLE_NAME);
	strcpy(running_process->proc_pointer->priority, "LOW");
	close(pfds_rq[1]);
	close(pfds_ret[0]);
	*request_fd = pfds_rq[0];
	*return_fd = pfds_ret[1];
}
```

```
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[])
{
	printf("start");
	int nproc;
	/* Two file descriptors for communication with the shell */
	static int request_fd, return_fd;
	nproc=argc-1;  // number of input processes
	number_of_processes=nproc; //input process, global variable to be used for pid's
	/* Create the shell. */
	sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd, &return_fd);
	/* TODO: add the shell to the scheduler's tasks */

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

	//nproc = 0; /* number of proccesses goes here */

	int i;
	for (i=1; i<=nproc; i++) {
		pid_t pid;
		pid=fork();
		char *argvs[] = { argv[i], NULL};
```

```
                char *newenviron[]={NULL};
                if (pid<0) {
                        perror("main: fork");
                        exit(1);
                }
                if (pid==0) {
                        raise(SIGSTOP);
                        execve( argv[i], argvs, newenviron);
                        perror("main: execve");
                        exit(1);
                }
                if (pid>0) {
                        proc_node *pn=(proc_node*)malloc(sizeof(proc_node));
                        process *pr=(process*)malloc(sizeof(process));
                        pr->id=i;
                        pr->pid=pid;
                        strcpy(pr->process_name, argv[i]);
                        strcpy(pr->priority, "LOW");
                        pn->next=running_process->next;
                        pn->proc_pointer=pr;
                        running_process->next=pn;
                        running_process=pn;

                }
        }

        running_process=running_process->next; //running process points to the last process we
added,point to the first one:FIFO

        /* 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) {                                       //We don't need this part:
There is always at least one  process to make running, the shell.
//              fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
//              exit(1);
//      }
//      if (nproc>0) {
                kill((running_process->proc_pointer)->pid,SIGCONT);  //wake up process which
entered the fifo queue fisrt, running_process is the one we added last
                alarm(SCHED_TQ_SEC);
//      }


        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;
}
```