ΣΧΟΛΗ
ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ
ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Λειτουργικά Συστήματα

ΑΝΑΦΟΡΑ 4ΗΣ ΕΡΓΑΣΤΗΡΙΑΚΗΣ ΑΣΚΗΣΗΣ

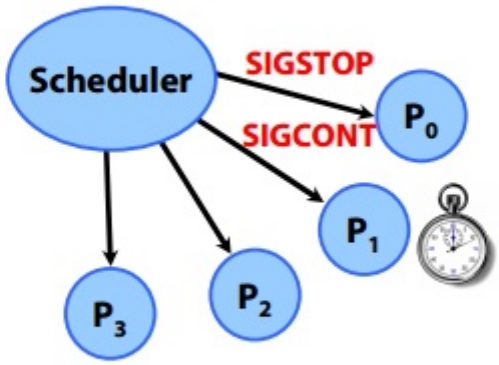
Φραγκάκη Μαρία-Ελένη
Α.Μ : 03112170

Πανταζόπουλος Γιώργος-Μιχαήλ
Α.Μ : 03112553

1) Υλοποίηση χρονοδρομολογητή κυκλικής επαναφοράς στο χώρο χρήστη :

Ζητείται η υλοποίηση ενός χρονοδρομολογητή κυκλικής επαναφοράς (round-robin). Ο χρονοδρομολογητής εκτελείται ως γονική διεργασία, στο χώρο χρήστη, κατανέμοντας τον υπολογιστικό χρόνο σε διεργασίες-παιδιά. Για τον έλεγχο των διεργασιών ο χρονοδρομολογητής θα 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 η διεργασία 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.



Σχήμα 1: Ο χρονοδρομολογητής σταματά την P_0 και ξεκινά την P_1.

Στην άσκηση αυτή ζητείται η υλοποίηση ενος χρονοδρομολογητή. Η διαδικασία – πατέρας (scheduler), δημιουργεί τόσες διαδικασίες παιδία, όσα το πλήθος των ορισμάτων. Τα παιδία αυτά οδήγουνται σε μια ουρά και δέχονται το σήμα SIGSTOP. Μόλις όλα τα παιδιά δεχτούν το σήμα, ο χροδρομολογητής πατέρας στέλνει το σήμα SIGCONT στη διαδικασία – παιδί που βρίσκεται πρώτο στην ουρά. Το παιδί αυτό εκτελεί τον κώδικα του μέχρι να εκπνεύσει το κβάντο χρόνου *tq* είτε να τερματίσει (δέχτηκε κάποιο εξωτερικό σήμα ή ολοκλήρωσε την εκτέλεσή του). Στην πρώτη περίπτωση, το παιδί θα μπεί στο τέλος της ουράς και θα περιμένει την σειρά του, ενώ στην δεύτερη θα αφαιρεθεί πλήρως από την ουρά. Στη συνέχεια ο χρονοδρομολογητής στέλνει το σήμα SIGCONT στο επόμενο παιδί στην ουρά. Η διαδικασία επαναλαμβάνεται μέχρι να τερματιστούν όλα τα παιδιά.

Ερωτήσεις :

1. Τι συμβαίνει αν το σήμα SIGALRM έρθει ενώ εκτελείται η συνάρτηση χειρισμού του σήματος SIGCHLD ή το αντίστροφο; Πώς αντιμετωπίζει ένας πραγματικός χρονοδρομολογητής χώρου πυρήνα ανάλογα ενδεχόμενα και πώς η δική σας υλοποίηση:

Το παραπάνω γεγονός φαίνεται να δημιουργεί πρόβλημα στο συγχρονισμό μεταξύ γονικής και θυγατρικής διαδικασίας. Εξετάζοντας των κώδικα της *install_signal_handlers()* παρατηρούμε κατά την εκτέλεση του προγράμματός μας, δημιουργείται μια μάσκα μεταξύ των δύο σημάτων. Πιο συγκεκριμένα το σήμα SIGALRM θα αγνοηθεί όσο εκτελείται η συνάρτηση χειρισμού του σήματος SIGCHLD και αντίστροφα. Αξίζει να σημειωθεί πως η εμβέλεια της μάσκας αυτής αφορά μόνο τα δύο αυτά σήματα. Ενδέχεται λοιπόν κατά την εκτέλεση της συνάρτησης χειρισμού SIGCHLD να σταλθεί κάποιο άλλο σήμα χωρίς το πρόγραμμα μας να μπορεί να το χειριστεί, με αποτέλεσμα να βρεθεί σε κατάσταση η οποία προσδιορίζεται από τον default χειριστή του εκάστοτε σήματος.

Ένας πραγματικός χρονοδρομολογητής είναι ευφυέστερος από τον χρονοδρομολογητή που υλοποιήσαμε στην άσκησή μας. Ένας τέτοιος χρονοδρομολογητής διαθέτει ένα struct για κάθε διαδικασία. Το struct αυτο περιέχει μία λίστα από τα σήματα που περιμένει να δεχτεί η εκάστοτε διαδικασία καθώς και μί αλίστα με τους handlers για τα αντίστοιχα σήματα. Όσο μια διαδικασία δέχεται σήματα, ο χρονοδρομολογητής ανάλογα με την λίστα της συγκεκριμένης διαδικασίας, τα τοποθετεί σε μία συγκεκριμένη σειρά προς εκτέλεση, και χρησιμοποιεί τους αντίστοιχους handlers για να χειριστεί τα εκάστοτε σήματα.

2.Κάθε φορά που ο χρονοδρομολογητής λαμβάνει σήμα SIGCHLD, σε ποια διεργασία- παιδί περιμένετε να αναφέρεται αυτό; Τι συμβαίνει αν λόγω εξωτερικού παράγοντα (π.χ. αποστολή SIGKILL) τερματιστεί αναπάντεχα μια οποιαδήποτε διεργασία- παιδί;

Η υλοποίησή μας βασίζεται σε μία ουρά. Στην κεφαλή της ουράς βρίσκεται η διεργασία-παιδί που τρέχει κάθε φορά. Αν η κατάσταση του παιδιού αλλάξει ο πατέρας χρονοδρομολογητής δέχεται ένα σήμα SIGCHLD. Συνεπώς το σήμα SIGCHLD αναφέρεται κάθε φορά στην διαδικασία παιδί που βρίσκεται στην κεφαλή της ουράς.

Όσον αφορά το σήμα SIGKILL έχουμε τις εξής περιπτώσεις :

α. Η διαδικασία-παιδί που βρίσκεται στην κεφαλή της ουράς δέχεται SIGKILL. Σε αυτήν την περίπτωση η τρέχουσα διαδικασία παιδί πεθαίνει και η επόμενη που θα τρέξει είναι η αμέσως επόμενη στην ουρά.

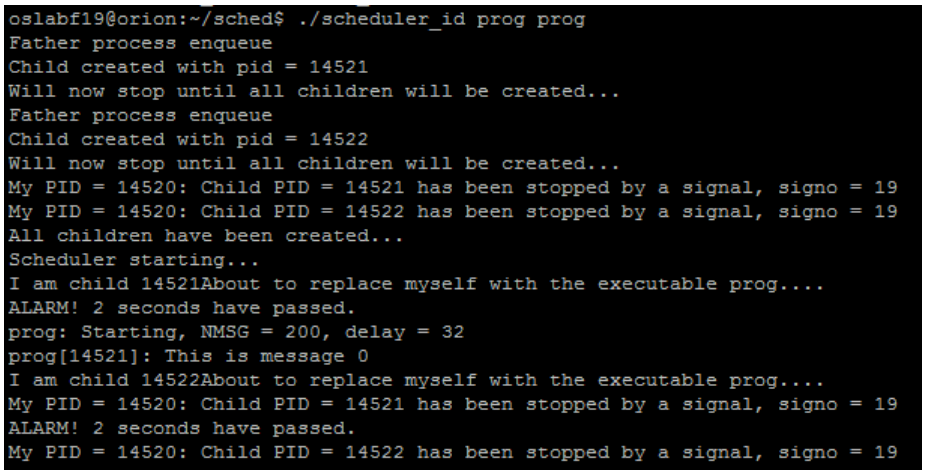
β. Μία ενδιάμεση διαδικασία στην ουρά δέχεται SIGKILL. Εδώ η συγκεκριμένη διαδικασία πεθαίνει και την θέση της παίρνει η αμέσως επόμενη στην ουρά , ενώ η διεργασία-παιδί στην κεφαλή συνεχίζει να εκτελείται.

γ. Η ουρά μας περιέχει μία μόνο διεργασία η οποία δέχεται SIGKILL. Εδώ η διεργασία θα πεθάνει και το πρόγραμμα θα τερματίσει.

3. Γιατί χρειάζεται ο χειρισμός δύο σημάτων για την υλοποίηση του χρονοδρομολογητή; θα μπορούσε ο χρονοδρομολογητής να χρησιμοποιεί μόνο το σήμα SIGALRM για να σταματά την τρέχουσα διεργασία και να ξεκινά την επόμενη; Τι ανεπιθύμητη συμπεριφορά θα μπορούσε να εμφανίζει μια τέτοια υλοποίηση; Υπόδειξη: Η παραλαβή του σήματος SIGCHLD εγγυάται ότι η τρέχουσα διεργασία έλαβε το σήμα SIGSTOP και έχει σταματήσει.

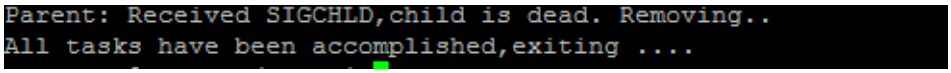
Καταρχάς εάν δεν χειριζόμαστε το σήμα SIGCHLD δεν θα μπορούσαμε να διαχειριστούμε τον 'θάνατο' μίας διεργασίας παιδιού. Η διεργασία χρονοδρομολογητής που έχουμε υλοποιήσει βρίσκεται στο επίπεδο χρήστη. Συνεπώς χρονοδρομολογείται και αυτή. Εάν λοιπόν στείλουμε SIGSTOP σε μία διεργασία μέσω SIGALRM υπάρχει το ενδεχόμενο ο χρονοδρομολογητής να στείλει SIGCONT στην επόμενη διεργασία για να εκτελεστεί, ενώ η προηγούμενη διεργασία παιδί να μην έχει λάβει ακόμη SIGSTOP. Έτσι μπορεί να έχουμε 2 διεργασίες να εκτελούνται παράλληλα,πράγμα που είναι ανεπιθύμητο.

Παρακάτω φαίνεται ενδεικτική έξοδος του προγράμματος :



Σχήμα 1. : Εκκίνηση προγράμματος.

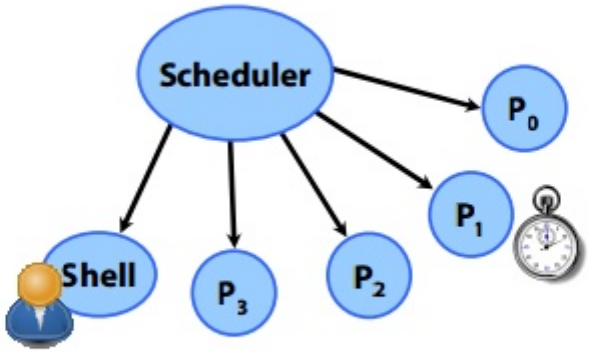
Σχήμα 2. : Περίπτωση τερματισμού παιδιού.



Σχήμα 3. : Τερματισμός προγράμματος.

2) Έλεγχος λειτουργίας χρονοδρομολογητή μέσω φλοιού

Ζητείται η επέκταση του χρονοδρομολογητή του προηγούμενου ερωτήματος, ώστε να υποστηρίζεται ο έλεγχος της λειτουργίας του μέσω προγράμματος-φλοιού. Ο χρήστης του συστήματος έχει τη δυνατότητα να ζητά δυναμική δημιουργία και τερματισμό διερ- γασιών, αλληλεπιδρώντας με το πρόγραμμα του φλοιού. Ο φλοιός δέχεται εντολές από το χρήστη, κατασκευάζει αιτήσεις κατάλληλης μορφής τις οποίες αποστέλλει προς τον χρονοδρομολογητή, λαμβάνει απαντήσεις που ενημερώνουν για την έκβαση της εκτέλεσής τους (επιτυχία / αποτυχία) και ενημερώνει για το αποτέλεσμά τους το χρήστη.



Σχήμα 2: Το πρόγραμμα-φλοιός ελέγχει τη λειτουργία του χρονοδρομολογητή

Ο φλοιός διαθέτει τέσσερις εντολές:
• Εντολή 'p': Ο χρονοδρομολογητής εκτυπώνει στην έξοδο λίστα με τις υπό εκτέλεση διεργασίες, στον οποίο φαίνεται ο σειριακός αριθμός id της διεργασίας, το PID και το όνομα της. Επιπλέον, επισημαίνεται η τρέχουσα διεργασία.
• Εντολή 'k': Δέχεται όρισμα το id μιας διεργασίας (προσοχή: όχι το PID) και ζητά από το χρονοδρομολογητή τον τερματισμό της.
• Εντολή 'e': Δέχεται όρισμα το όνομα ενός εκτελέσιμου στον τρέχοντα κατάλογο, π.χ. prog2 και ζητά τη δημιουργία μιας νέας διεργασίας από τον χρονοδρομολο- γητή, στην οποία θα τρέχει αυτό το εκτελέσιμο.
• Εντολή 'q': Ο φλοιός τερματίζει τη λειτουργία του.

Για την επέκταση του χρονοδρομολογητή, εντάξαμε στην ουρά μας τον φλοιό ως μια διαδικασία προς χρονοδρομολόγηση. Ο χρήστης πλέον μπορεί να επέμβει στην ουρά των διαδικασιών προς χρονοδρομολόγηση, μπορεί να δημιουργήσει νέες, να σκοτώσει κάποια παλιά, αλλά όχι να αλλάξει την σειρά με την οποία εκτελούνται οι διαδικασίες. Για να γίνει αυτό έχουν ενσωματωθεί οι παραπάνω εντολές του φλοιού. Όταν ο φλοίος βρίσκεται στην κεφαλή της ουράς (δηλαδή εκτελείται ο κώδικάς του), αναμένει μία συγκεκριμένη εντολή μέσω του command-line. Γίνεται μια αντιστοίχιση του αιτήματος μέσω ανάλογων συναρτήσεων και εξυπηρετείται η αντίστοιχη εντολή.

Ερωτήσεις:

1. Όταν και ο φλοιός υφίσταται χρονοδρομολόγηση, ποια εμφανίζεται πάντοτε ως τρέχουσα διεργασία στη λίστα διεργασιών (εντολή 'p'); Θα μπορούσε να μη συμβαίνει αυτό; Γιατί;

Στην υλοποίησή μας τυπώνεται πρώτα πάντα ο φλοιός. Σε περίπτωση που δώσουμε την εντολή λίγο πριν εκπνεύσει το κβάντο χρόνου, μπορεί ο χρονοδρομολογητής να έχει ήδη προλάβει να δείξει στην επόμενη διεργασία,οπότε και να εμφανιστεί αυτή. Αυτό οφείλεται στο γεγονός ότι η διαχείριση των σημάτων δεν είναι απόλυτα σύγχρονη .

2. Γιατί είναι αναγκαίο να συμπεριλάβετε κλήσεις signals_disable(), _enable() γύρω από την συνάρτηση υλοποίησης αιτήσεων του φλοιού; Υπόδειξη: Η συνάρτηση υλοποίησης αιτήσεων του φλοιού μεταβάλλει δομές όπως η ουρά εκτέλεσης των διεργασιών.

Ο λόγος που χρειαζόμαστε τις signals_enable() και signals_disable() σε οποιαδήποτε κλήση εξωτερικής διεργασίας είναι για να διασφαλίζεται η δομή της ουράς. Οι διεργασίες του φλοιού δεν επηρεάζουν όλες την δομή της ουράς, όπως για παράδειγμα η 'p' (print). Ωστόσο, αιτήσεις όπως η 'k' (kill) ή 'e' (create new task) μπορούν να επηρεάσουν την ουρά , γιαυτό πρέπει να εξασφαλίσουμε ότι η σειρά με την οποία εκτελούνται οι διεργασίες διατηρείται.

Παραθέτουμε και εδώ ενδεικτική έξοδο του προγράμματός μας (./scheduler-shell prog prog).

```
My PID = 15428: Child PID = 15431 has been stopped by a signal, signo = 19
name : shell | id =  0 | pid = 15429
name : prog | id =  1 | pid = 15430
name : prog | id =  2 | pid = 15431
```

Σχήμα 4. Εντολή 'p' .

```
name : shell | id =  0 | pid = 15433
name : prog | id =  2 | pid = 15435
name : prog | id =  3 | pid = 15881
```

Σχήμα 5. Εντολή 'p' ύστερα από την εκτέλεση της 'k 1' και της 'e prog'

3) Υλοποίηση προτεραιοτήτων στο χρονοδρομολογητή

Ζητείται η επέκταση του χρονοδρομολογητή του προηγούμενου ερωτήματος, ώστε να υποστηρίζονται δύο κλάσεις προτεραιότητας: HIGH και LOW. Ο αλγόριθμος χρονοδρομολόγησης αλλάζει ως εξής: αν υπάρχουν διεργασίες προτεραιότητας HIGH εκτελούνται μόνο αυτές χρησιμοποιώντας κυκλική επαναφορά (round-robin). Σε αντίθετη περίπτωση, χρονοδρομολογούνται οι LOW διεργασίες χρησιμοποιώντας κυκλική επαναφορά. Όλες οι διεργασίες δημιουργούνται με LOW προτεραιότητα. Η αλλαγή της προτεραιότητας μιας διεργασίες θα πραγματοποιείται με τις παρακάτω εντολές φλοιού:

• 'h' ('l') Δέχεται όρισμα το id μιας διεργασίας και θέτει την προτεραιότητα της σε HIGH (LOW).

Τέλος επεκτείνουμε τις δυνατότητες του φλοιού. Πλέον ο χρήστης μπορεί να επέμβει πλήρως στην ουρά, καθώς μπορεί να αλλάξει την προτεραιότητα των διαδικασιών προς χρονοδρομολόγηση. Για τον σκοπό αυτό έχει ενσωματωθεί η αντίστοιχη συνάρτηση εξυπηρέτησης της εντολής προτεραιότητας στον κώδικα του φλοιού.

Τέλος ενσωματώνουμε ενδεικτική έξοδο για την εντολή h και l αντίστοιχα :

```
Shell: issuing request...
Shell: receiving request return value...
Shell> ALARM! 2 seconds have passes.
ALARM! 2 seconds have passes.
ALARM! 2 seconds have passes.
ALARM! 2 seconds have passes.
ALARM! 2 seconds have passes.
ALARM! 2 seconds have passes.
ALARM! 2 seconds have passes.
ALARM! 2 seconds have passes.
```

Σχήμα 6 : Προτεραιότητα στον φλοιό.

```
Shell> ALARM! 2 seconds have passes.
p
Shell: issuing request...
Shell: receiving request return value...
Shell> name : shell | id =  0 | pid = 15888 | prio = 1ALARM! 2 seconds have passes.
ALARM! 2 seconds have passes.
```

Σχήμα 7 : Εντολή 'p' ,προτεραιότητα μόνο στον φλοιό.

```
h ALARM! 2 seconds have passes.
1
Shell: issuing request...
Shell: receiving request return value...
Shell> ALARM! 2 seconds have passes.
My PID = 15887: Child PID = 15888 has been stopped by a signal, signo = 19
prog[15889]: This is message 32
prog[15889]: This is message 33
prog[15889]: This is message 34
prog[15889]: This is message 35
prog[15889]: This is message 36
ALARM! 2 seconds have passes.
My PID = 15887: Child PID = 15889 has been stopped by a signal, signo = 19
ALARM! 2 seconds have passes.
My PID = 15887: Child PID = 15888 has been stopped by a signal, signo = 19
prog[15889]: This is message 37
prog[15889]: This is message 38
prog[15889]: This is message 39
p
prog[15889]: This is message 40
ALARM! 2 seconds have passes.
Shell: issuing request...
Shell: receiving request return value...
Shell> My PID = 15887: Child PID = 15889 has been stopped by a signal, signo = 19
name : shell | id =  0 | pid = 15888 | prio = 1name : prog | id =  1 | pid = 15889 | prio = 1ALARM! 2 seconds have passes.
My PID = 15887: Child PID = 15888 has been stopped by a signal, signo = 19
prog[15889]: This is message 41
prog[15889]: This is message 42
prog[15889]: This is message 43
prog[15889]: This is message 44
prog[15889]: This is message 45
```

Σχήμα 8. : Εντολή 'p', προτεραιότητα στον φλοιό και στην διαδικασία με id = 1.

```
ALARM! 2 seconds have passes.
My PID = 15887: Child PID = 15890 has been stopped by a signal, signo = 19
1 0
Shell: issuing request...
Shell: receiving request return value...
Shell> ALARM! 2 seconds have passes.
My PID = 15887: Child PID = 15888 has been stopped by a signal, signo = 19
prog[15890]: This is message 43
prog[15890]: This is message 44
prog[15890]: This is message 45
prog[15890]: This is message 46
prog[15890]: This is message 47
ALARM! 2 seconds have passes.
prog[15890]: This is message 48
prog[15890]: This is message 49
prog[15890]: This is message 50
prog[15890]: This is message 51
prog[15890]: This is message 52
prog[15890]: This is message 53
ALARM! 2 seconds have passes.
prog[15890]: This is message 54
prog[15890]: This is message 55
prog[15890]: This is message 56
prog[15890]: This is message 57
prog[15890]: This is message 58
ALARM! 2 seconds have passes.
prog[15890]: This is message 59
prog[15890]: This is message 60
prog[15890]: This is message 61
prog[15890]: This is message 62
prog[15890]: This is message 63
ALARM! 2 seconds have passes.
prog[15890]: This is message 64
prog[15890]: This is message 65
prog[15890]: This is message 66
prog[15890]: This is message 67
prog[15890]: This is message 68
prog[15890]: This is message 69
ALARM! 2 seconds have passes.
prog[15890]: This is message 70
prog[15890]: This is message 71
prog[15890]: This is message 72
prog[15890]: This is message 73
prog[15890]: This is message 74
ALARM! 2 seconds have passes.
prog[15890]: This is message 75
prog[15890]: This is message 76
prog[15890]: This is message 77
prog[15890]: This is message 78
prog[15890]: This is message 79
ALARM! 2 seconds have passes.
prog[15890]: This is message 80
prog[15890]: This is message 81
prog[15890]: This is message 82
prog[15890]: This is message 83
prog[15890]: This is message 84
```

Σχήμα 9.: Προτεραιότητα μόνο στην διαδικασία με pid = 15890.

Ερωτήσεις:

1) Περιγράψτε ένα σενάριο δημιουργίας λιμοκτονίας.

Στην υλοποίηση μας μπορούμε να διακρίνουμε τα εξής σενάρια λιμοκτονίας :

- Ένα πιθανό σενάριο λιμοκτονίας είναι ο φλοιός να δώσει προτεραιότητα 'HIGH' σε μία διεργασία κι αυτή να τερματίσει προτού προλάβει ο φλοιός να δώσει HIGH σε άλλη διεργασία οπότε να μην εκτελεστεί καμία άλλη.
- Η διαδικασία η οποία τροφοδοτείται με υψηλή προτεραιότητα, περιέχει έναν ατέρμονο βρόχο. Παράλληλα τροφοδοτούμε τον φλοιό με χαμηλή προτεραιότητα. Συνεπώς η παραπάνω διαδικασία θα εκτελείται για πάντα χωρίς να επιτρέπει την εκτέλεση καμίας άλλης ακόμα και του φλοιού.

Παράρτημα : Επίδειξη κώδικα.

Άσκηση 1 :

```
#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 */


struct child{
        pid_t mypid;
        int myid ;
        struct child *next;
};
typedef struct child child;

child *head;
child *tail;
int id;


void children(const char *name){
        pid_t mypid = getpid();
        printf("I am child %ld",(long)mypid);

        char executable[TASK_NAME_SZ];
        strcpy(executable,name);

        char *newargv[] = { executable, NULL, NULL, NULL };
        char *newenviron[] = { NULL };

        printf("About to replace myself with the executable %s....\n",executable);
        sleep(2);

        execve(executable, newargv, newenviron);

        printf("Execve failed,exiting...\n");
        perror("execve");
        exit(1);

}
```

```
void enqueue(pid_t pid, int flag){

	//Queue is empty
	if(head == NULL){
		head = malloc(sizeof(child));
		head->mypid = pid;
		if(flag == 1){
			head->myid = id;
			id++;
		}
		head->next = NULL;
		tail = head;
	}
	else {
		child *new = malloc(sizeof(child));
		new->mypid = pid;
		if (flag == 1){
			new->myid = id;
			id++;
		}
		new->next = NULL;
		tail->next = new;
		tail = new;

	}
}
pid_t dequeue(){
	child *temp;
	pid_t head_pid;
	head_pid = head->mypid;
	temp = head;
	head = head->next;
	free(temp);
	return head_pid;
}

int remove_dead_child(pid_t dead_child){
/*
 *3 possible scenarios :
 *1)The child died and queue is empty => All tasks have been accomplished.
 *2)The child, first in the queue, died => Remove it from queue and renew timer on the next one.
 *3)A child between head and tail died => Simply remove it from queue.
 */

	child *temp;
	child *pretail;
//First scenario :
	if(head == tail){
		free(head);
		return 1;
	}
//Second scenario :
	if(head->mypid == dead_child){
		temp = head;
		head = head->next;
```

```
                free(temp);
                return 2;
        }
        temp = head;
        pretail = head;
//Third scenario :
        while (temp != NULL){

        if (temp->mypid == dead_child){
        if (temp->next == NULL){
                tail = pretail;
                        tail->next = NULL;
                        free(temp);
                return 0;
        }
        else{
                pretail->next = temp->next;
                free(temp);
                return 0;
        }
      }
      else{
         pretail = temp;
         temp = temp->next;
      }
   }
        return 0;
}

/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum)
{
        //assert(0 && "Please fill me!");//
        if (signum != SIGALRM ) {
                fprintf(stderr, " Internal error: Called for signum %d,not SIGALRM\n",signum);
                exit(1);
        }

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

        pid_t remove_from_queue;

        remove_from_queue = dequeue();
        enqueue(remove_from_queue,0);

        kill(remove_from_queue,SIGSTOP);
        kill(head->mypid,SIGCONT);

        if (alarm(SCHED_TQ_SEC)<0){
                perror("alarm");
                exit(1);
        }
```

```
}

/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
	//assert(0 && "Please fill me!");//
	pid_t p;
	int status;

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

	}

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

		}
		if (p==0)  break;

		explain_wait_status(p,status);

		if(WIFEXITED(status) || WIFSIGNALED(status)){
			printf("Parent: Received SIGCHLD,child is dead. Removing..\n");
		      int result = remove_dead_child(p);

			if(result == 1){
				printf("All tasks have been accomplished,exiting ....\n");
				exit(1);
			}
			else if(result == 2){
				kill(head->mypid,SIGCONT);
				printf("Refreshing alarm for next child..\n");

				if(alarm(SCHED_TQ_SEC)<0){
				 perror("alarm");
					exit(1);
			}
			}

			if(WIFSTOPPED(status)){
				printf("Parent: Child has been stopped. Moving right along..\n");
			}
		}
	}
}

/* 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[])
{
	int i,nproc;
	pid_t p;
	tail = NULL;
	head = NULL;
	id = 0;
	/*
	 * For each of argv[1] to argv[argc - 1],
	 * create a new child process, add it to the process list.
	 */

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

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

	/* Install SIGALRM and SIGCHLD handlers. */

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

```
            exit(1);
        }

        for(i=0; i<nproc; i++){
                p = fork();
                if(p<0){
                        perror("fork");
                        exit(1);
                }
                else if (p == 0){
                /*
                 *Child
                 */
                        printf("Child created with pid = %ld \n" ,(long)getpid());
                        printf("Will now stop until all children will be created...\n");
                        raise(SIGSTOP);
                        children(argv[i+1]);
                }
                else {
                /*
                 *Father
                 */
                        printf("Father process enqueue \n");
                        enqueue(p,1);
                }
        }

        wait_for_ready_children(nproc);
        install_signal_handlers();

        printf("All children have been created...\n");
        printf("Scheduler starting...\n");
        kill(head->mypid,SIGCONT);

        alarm(SCHED_TQ_SEC);

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

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

        return 1;
}
```

Άσκηση 2 :

```
#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 */

/*Struct Process ID*/

struct PT{
        int myid;
        pid_t mypid;
        char name[TASK_NAME_SZ];
        struct PT *next;
};
typedef struct PT ProcessTable ;

/*id for every process*/
int id;
/*Head&Tail of queue*/
ProcessTable *head;
ProcessTable *tail;

void children(const char *name){
        pid_t mypid = getpid();
        printf("I am child %ld",(long)mypid);

        char executable[TASK_NAME_SZ];
        strcpy(executable,name);

        char *newargv[] = { executable, NULL, NULL, NULL};
        char *newenviron[] = { NULL };

        printf("About to replace myself with the executable %s... \n", executable);

        sleep(2);
        execve(executable,newargv,newenviron);

        printf("Execve failed...\n");
        perror("execve");
        exit(1);

}
```

```
void enqueue(ProcessTable node, int flag){
	if(head == NULL){
		head = malloc(sizeof(ProcessTable));
		head->mypid = node.mypid;
		if(flag == 1){
			head->myid = id;
			id++;
		}
		strcpy(head->name,node.name);
		head->next = NULL;
		tail = head;
	}
	else{
		ProcessTable *new = malloc(sizeof(ProcessTable));
		new->mypid = node.mypid;
		if(flag == 1){
			new->myid = id;
			id++;
		}
		else{
			new->myid = node.myid;
		}
		strcpy(new->name,node.name);
		new->next = NULL;
		tail->next = new;
		tail = new;
	}
}

ProcessTable dequeue(){
	ProcessTable *temp,dequeued;

	dequeued.mypid = head->mypid;
	dequeued.myid = head->myid;
	strcpy(dequeued.name,head->name);
	temp = head;
	head = head->next;
	free(temp);
	return dequeued;
}

int remove_dead_child(pid_t dead_process){
	ProcessTable *temp;
	ProcessTable *pretail;
//First scenario:
	if(head == tail){
		free(head);
		return 1;
	}
//Second scenario:
	if(head->mypid == dead_process){
		temp = head;
		head = head->next;
		free(temp);
		return 2;
	}
```

```
	temp = head;
	pretail = head;
//Third scenario:
	while(temp!= NULL){
		if(temp->mypid == dead_process){
			if(temp->next == NULL){
				tail = pretail;
				tail->next = NULL;
				free(temp);
				return 0;

			}
			else{
				pretail->next = temp->next;
				free(temp);
				return 0;
			}
		}
		else{
			pretail = temp;
			temp = temp->next;

		}
	}
	return 0;
}

/*: Print a list of all tasks currently being scheduled.  */
static void
sched_print_tasks(void)
{
	ProcessTable *temp;
	temp = head;


	while(temp != NULL){
		printf("name : %s | id =  %d | pid = %d \n", temp->name, temp->myid, temp->mypid);
		temp = temp -> 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 my_id)
{
	//assert(0 && "Please fill me!");
	ProcessTable *temp;
	temp = head;
	while(temp != NULL){
		if((temp->myid) == my_id){
			kill((temp->mypid),SIGTERM);
			break;
		}
		temp = temp -> next;
```

```
        }
        return -ENOSYS;
}


/* Create a new task.  */
static void
sched_create_task(char *executable)
{
        //assert(0 && "Please fill me!");//

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

        if (p==0){
                raise(SIGSTOP);
                children(executable);
        }
        ProcessTable process;
        process.mypid = p;
        strcpy(process.name,executable);
        enqueue(process,1);
}

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

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

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

                default:
                        return -ENOSYS;
        }
}

/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum)
{
        //assert(0 && "Please fill me!");//
        if(signum!= SIGALRM){
                fprintf(stderr,"Internal error: Called for signum %d,not SIGALRM",signum);
```

```
                exit(1);

        }

        printf("ALARM! %d seconds have passes.\n",SCHED_TQ_SEC);

        ProcessTable remove_from_queue;

        remove_from_queue = dequeue();
        enqueue(remove_from_queue,0);

        kill(remove_from_queue.mypid,SIGSTOP);
        kill(head->mypid,SIGCONT);

        if(alarm(SCHED_TQ_SEC)<0){
                perror("alarm");
                exit(1);
        }
}

/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
        pid_t p;
        int status;

        if (signum!= SIGCHLD){
                fprintf(stderr,"Internal error: Called for signum %d, not SIGCHLD\n",signum);
                exit(1);
         }
        for(;;){
                p = waitpid(-1, &status, WUNTRACED | WNOHANG);
                if (p<0) {
                        perror("waitpid");
                        exit(1);
                }

                if (p==0)  break;
                explain_wait_status(p,status);


                if (WIFEXITED(status) || WIFSIGNALED(status)){
                        printf("Parent: Received SIGCHLD,child is dead.Removing...\n");
                        int result = remove_dead_child(p);

                        if(result == 1){
                                printf("All tasks have been accomplished,exiting...\n");
                                exit(1);
                        }

                        else if (result == 2){
                                kill(head->mypid,SIGCONT);
                                printf("Refreshing alarm for next child...\n");
                                if (alarm(SCHED_TQ_SEC)<0){
```

```
                                        perror("alarm");
                                        exit(1);

                                }

                        }

                        if (WIFSTOPPED(status)){
                                printf("Parent: Child has been stopped. Moving right along..\n");
                        }
                }
        }
}
/* 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 p = p_shell store it in queue */
	close(pfds_rq[1]);
	close(pfds_ret[0]);
	*request_fd = pfds_rq[0];
	*return_fd = pfds_ret[1];

	//Store in queue
	ProcessTable process;
	process.mypid = p;
	strcpy(process.name, SHELL_EXECUTABLE_NAME);
	enqueue(process,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[])
{
        int i,nproc;
        id = 0;
        pid_t p;
        tail = NULL;
        head = NULL;

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

        /* Create the shell. */
        sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd, &return_fd);
        /*  add the shell to the scheduler's tasks->mallon egine */

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


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

        for(i=0; i<nproc; i++){
                p = fork();
                if(p < 0){
                        perror("fork");
                        exit(1);
                }
                else if(p == 0){
                        //Child
                        raise(SIGSTOP);
                        children(argv[i+1]);
                }
                else {
                        printf("Father process enqueue..\n");
                        ProcessTable process;
                        process.mypid = p;
                        strcpy(process.name,argv[i+1]);
                        enqueue(process,1);
                }
        }

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

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

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

```
        printf("All children are ready,start schedule....\n");
        kill(head->mypid,SIGCONT);
        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 3 :

```
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 */

/*Struct Process ID*/

struct PT{
        int myid;
        int mypriority;
        pid_t mypid;
        char name[TASK_NAME_SZ];
        struct PT *next;
};
typedef struct PT ProcessTable ;

/*id for every process*/
int id;
/*Head&Tail of queue*/
ProcessTable *head;
ProcessTable *tail;

void printQ(){
        ProcessTable *temp;
        temp = head;
        printf("-----------Queue-----------------------\n");
        while(temp != NULL){
                printf("name = %s, pid = %d, prio = %d \n",temp->name,temp->mypid,temp->mypriority);
                temp = temp -> next;
        }
        printf("---------------------------------------\n");
}


void children(const char *name){
        pid_t mypid = getpid();
        printf("I am child %ld",(long)mypid);

        char executable[TASK_NAME_SZ];
        strcpy(executable,name);
```

```
        char *newargv[] = { executable, NULL, NULL, NULL};
        char *newenviron[] = { NULL };

        printf("About to replace myself with the executable %s... \n", executable);

        sleep(2);
        execve(executable,newargv,newenviron);

        printf("Execve failed...\n");
        perror("execve");
        exit(1);

}

void enqueue(ProcessTable node ,int flag){
        if(head == NULL){
                head = (ProcessTable *) malloc(sizeof(ProcessTable));
                head->mypid = node.mypid;
                head->mypriority = node.mypriority;
                if(flag ==1){
                        head->myid = id;
                        id++;
                }
                printf("priority on enqueue; %d\n", node.mypriority);
                strcpy(head->name,node.name);
                head->next = NULL;
                tail = head;
        }
        else{
                ProcessTable *new = (ProcessTable*) malloc(sizeof(ProcessTable));
                new->mypid = node.mypid;
                if(flag == 1){
                        new->myid = id;
                        id++;
                }
                else {
                        new->myid = node.myid;
                }
                new->mypriority = node.mypriority;
                strcpy(new->name,node.name);
                new->next = NULL;
                tail->next = new;
                tail = new;
        }
}

ProcessTable dequeue(){
        ProcessTable *temp,dequeued;

        dequeued.mypid = head->mypid;
        dequeued.mypriority = head->mypriority;
        dequeued.myid = head->myid;
        strcpy(dequeued.name,head->name);
        temp = head;
        head = head->next;
        free(temp);
        return dequeued;
```

```
}

int IsItHigh(){
	ProcessTable *temp;
	temp = head;

	while(temp != NULL){
		if((temp->mypriority) == 1)
			return 1;
		else
			temp = temp->next;
	}
	return 0;
}

void FindNext(){
	ProcessTable remove_from_queue;
	ProcessTable *temp;
	int check;
	int count = 0;
	int i,flag;
	flag = 0;
	check = IsItHigh();

	if(check == 0){
		remove_from_queue = dequeue();
		kill(remove_from_queue.mypid,SIGSTOP);
		enqueue(remove_from_queue,0);
		kill(head->mypid,SIGCONT);
	}
	else{
		remove_from_queue = dequeue();
		kill(remove_from_queue.mypid,SIGSTOP);
		enqueue(remove_from_queue,0);

		temp = head;

		while(temp != NULL){
			if((temp->mypriority) == 1){
				flag = 1;
				break;
			}
			else{
				temp = temp ->next;
				count++;
			}
		}
		if(flag == 1) {
			for(i=0; i<count; i++){
				remove_from_queue = dequeue();
				enqueue(remove_from_queue,0);
			}
		}

		kill(head->mypid,SIGCONT);
	}
}
```

```
int remove_dead_child(pid_t dead_process){
	ProcessTable *temp;
	ProcessTable *pretail;
//First scenario:
	if(head == tail){
		free(head);
		return 1;
	}
//Second scenario:
	if(head->mypid == dead_process){
		temp = head;
		head = head->next;
		free(temp);
		return 2;
	}
	temp = head;
	pretail = head;
//Third scenario:
	while(temp!= NULL){
		if(temp->mypid == dead_process){
			if(temp->next == NULL){
				tail = pretail;
				tail->next = NULL;
				free(temp);
				return 0;

			}
			else{
				pretail->next = temp->next;
				free(temp);
				return 0;
			}
		}
		else{
			pretail = temp;
			temp = temp->next;

		}
	}
	return 0;
}

/*: Print a list of all tasks currently being scheduled.  */
static void
sched_print_tasks(void)
{
	int check;
	ProcessTable *temp;
	temp = head;
	check = IsItHigh();

	if(check == 0){
		while(temp != NULL){
			printf("name : %s | id =  %d | pid = %d | prio = %d \n", temp->name, temp->myid,
temp->mypid, temp->mypriority);
```

```
                        temp = temp -> next;
                }
        }
        else{
                while(temp != NULL){
                        if((temp->mypriority) == 1){
                                printf("name : %s | id =  %d | pid = %d | prio = %d", temp->name, temp-
>myid, temp->mypid, temp->mypriority);

                        }
                        temp = temp->next;
                }
        }
}

/*: Send SIGKILL to a task determined by the value of its
 * scheduler-specific id.
 */
static int
sched_kill_task_by_id(int my_id)
{
        //assert(0 && "Please fill me!");
        ProcessTable *temp;
        temp = head;
        while(temp != NULL){
                if((temp->myid) == my_id){
                        kill((temp->mypid),SIGTERM);
                        break;
                }
                temp = temp -> next;
        }

        return -ENOSYS;
}


/* Create a new task.  */
static void
sched_create_task(char *executable)
{
        //assert(0 && "Please fill me!");//

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

        if (p==0){
                raise(SIGSTOP);
                children(executable);
        }
        ProcessTable process;
        process.mypid = p;
        strcpy(process.name,executable);
        enqueue(process,1);
}
```

```
static void
sched_priority(int my_id,int my_priority){

	ProcessTable *temp;
	temp = head;

	while(temp != NULL){
		if((temp->myid) == my_id){
			temp->mypriority = my_priority;
			break;
		}
		else
			temp = temp->next;

	}

}

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

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

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

			sched_priority(rq->task_arg,1);
			return 0;
		case REQ_LOW_TASK:
			sched_priority(rq->task_arg,0);
			return 0;
		default:
			return -ENOSYS;
	}
}

/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum)
{
	//assert(0 && "Please fill me!");//
	if(signum!= SIGALRM){
		fprintf(stderr,"Internal error: Called for signum %d,not SIGALRM",signum);
		exit(1);
```

```
	}

	printf("ALARM! %d seconds have passes.\n",SCHED_TQ_SEC);

	FindNext();
	if(alarm(SCHED_TQ_SEC)<0){
		perror("alarm");
		exit(1);
	}
}

/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
	pid_t p;
	int status;

	if (signum!= SIGCHLD){
		fprintf(stderr,"Internal error: Called for signum %d, not SIGCHLD\n",signum);
		exit(1);
	 }
	for(;;){
		p = waitpid(-1, &status, WUNTRACED | WNOHANG);
		if (p<0) {
			perror("waitpid");
			exit(1);
		}

		if (p==0)  break;
		explain_wait_status(p,status);


		if (WIFEXITED(status) || WIFSIGNALED(status)){
			printf("Parent: Received SIGCHLD,child is dead.Removing...\n");
			int result = remove_dead_child(p);

			if(result == 1){
				printf("All tasks have been accomplished,exiting...\n");
				exit(1);
			}

			else if (result == 2){
				FindNext();
				printf("Refreshing alarm for next child...\n");
				if (alarm(SCHED_TQ_SEC)<0){
					perror("alarm");
					exit(1);

				}

			}

			if (WIFSTOPPED(status)){
				printf("Parent: Child has been stopped. Moving right along..\n");
```

```
                        }
                }
        }
}
/* 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 p = p_shell store it in queue */
	close(pfds_rq[1]);
	close(pfds_ret[0]);
	*request_fd = pfds_rq[0];
	*return_fd = pfds_ret[1];

	//Store in queue
	ProcessTable process;
	process.mypid = p;
	process.mypriority = 0;
	strcpy(process.name, SHELL_EXECUTABLE_NAME);
	enqueue(process,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[])
{
	int i,nproc;
	id = 0;
	pid_t p;
```

```
tail = NULL;
head = NULL;

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

/* Create the shell. */
sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd, &return_fd);
/*  add the shell to the scheduler's tasks->mallon egine */

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


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

for(i=0; i<nproc; i++){
        p = fork();
        if(p < 0){
                perror("fork");
                exit(1);
        }
        else if(p == 0){
                //Child
                raise(SIGSTOP);
                children(argv[i+1]);
        }
        else {
                printf("Father process enqueue..\n");
                ProcessTable process;
                process.mypid = p;
                process.mypriority = 0;
                strcpy(process.name,argv[i+1]);
                enqueue(process,1);
        }
}

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

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

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

printf("All children are ready,start schedule....\n");
kill(head->mypid,SIGCONT);
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;
}
```