2η Άσκηση στα Λειτουργικά Συστήματα

6ο Εξάμηνο, Ακαδημαϊκή περίοδος 2020 – 2021

Ονοματεπώνυμο	Αριθμός Μητρώου
Στεφανάκης Γεώργιος	el18436
Τζανακάκης Αλέξανδρος	el18431

1.1 Δημιουργία δεδομένου δέντρου διεργασιών

Ζητείται η κατασκευή προγράμματος που να δημιουργεί το δέντρο διεργασιών που περιγράφεται στην εκφώνηση της άσκησης. Οι διεργασίες δημιουργούνται και περιμένουν τον τερματισμό των διεργασιών – παιδιών τους ενώ τα φύλλα του δέντρου εκτελούν την εντολή `sleep()`. Αυτό γίνεται ώστε οι διεργασίες να διατηρηθούν ενεργές για ένα χρονικό διάστημα, αρκετό ώστε να μπορέσουμε να παρατηρήσουμε το δέντρο μέσω της `show_pstree()`. Κατά τη διάρκεια της εκτέλεσης του προγράμματος τυπώνονται και τα ανάλογα μηνύματα.

Η έξοδος του πρώτου προγράμματος με χρήση της `show_pstree(pid)` στο σώμα της `main` είναι η εξής:

```
$ ./ask1-given-tree
A begins...
A forks B...
A forks C...
B begins...
B forks D...
A waits for B and C...
B waiting for D...
C begins...
C sleeping...
D begins...
D sleeping...


A(5391)─┬─B(5392)───D(5394)
        └─C(5393)


D exiting...
C exiting...
My PID = 5392: Child PID = 5394 terminated normally, exit status = 13
My PID = 5391: Child PID = 5393 terminated normally, exit status = 17
B exiting...
My PID = 5391: Child PID = 5392 terminated normally, exit status = 19
A exiting...
My PID = 5390: Child PID = 5391 terminated normally, exit status = 16
$
```

Επιπλέον, παραθέτουμε και την έξοδο του πρώτου προγράμματος με χρήση της εντολής `show_pstree(getpid())` στο σώμα της main.

```
$ ./ask1-given-tree
A begins...
A forks B...
A forks C...
B begins...
B forks D...
A waits for B and C...
B waiting for D...
C begins...
C sleeping...
D begins...
D sleeping...


ask1-given-tree(5949)─┬─A(5950)─┬─B(5951)──D(5953)
                      │         └─C(5952)
                      └─sh(5954)──pstree(5955)


C exiting...
D exiting...
My PID = 5951: Child PID = 5953 terminated normally, exit status = 13
My PID = 5950: Child PID = 5952 terminated normally, exit status = 17
B exiting...
My PID = 5950: Child PID = 5951 terminated normally, exit status = 19
A exiting...
My PID = 5949: Child PID = 5950 terminated normally, exit status = 16
$
```

1) Εάν τερματίσουμε πρόωρα τη διεργασία Α χρησιμοποιώντας την εντολή `kill -KILL 5949`, όλες οι διεργασίες – παιδιά της Α, δηλαδή όλο το δέντρο διεργασιών που κρέμεται από αυτήν, θα μείνουν «ορφανά» και θα υιοθετηθούν από την init, η οποία είναι διεργασία υπ' αριθμόν 1 και η πρώτη που δημιουργεί το λειτουργικό σύστημα κατά την εκκίνησή του. Η init κάνει συνεχώς wait μέχρι να ολοκληρωθούν τα παιδιά της, συνεπώς θα περιμένει μέχρι να τερματίσουν όλα τα παιδιά της διεργασίας Α.
2) Η κλήση συστήματος `getpid()` μας επιστρέφει το process ID της εκτελούμενης διεργασίας. Άρα, όπως επαληθεύεται και από τις παραπάνω εξόδους του προγράμματος, με την εντολή `show_pstree(pid)` θα τυπωθεί το δέντρο διεργασιών που έχει ως ρίζα τη διεργασία Α, εκείνη δηλαδή που δημιουργήσαμε κάνοντας `fork()` στη `main()`, ενώ με την εντολή `show_pstree(getpid())` θα τυπωθεί το δέντρο που έχει ρίζα το αρχικό process το οποίο κατασκεύασε ο φλοιός κατά την εκκίνηση του προγράμματος. Το αποτέλεσμα που παίρνουμε και στις δύο περιπτώσεις είναι αναμενόμενο και ιδιαίτερο ενδιαφέρον παρουσιάζει το περαιτέρω κλαδί διεργασιών που εμφανίζεται στη δεύτερη έξοδο το οποίο έχει σκοπό την κατασκευή μίας διεργασίας φλοιού που θα εκτελέσει την κλήση συστήματος `pstree` του linux η οποία τυπώνει ένα δέντρο διεργασιών. Προφανώς, θα τυπωθεί επιπλέον και το δέντρο διεργασιών που κατασκευάσαμε στο πρόγραμμα.
3) Ένα υπολογιστικό σύστημα προκειμένου να αποφύγει την δημιουργία fork bombs (ή rabbit virus ή wabbit), δηλαδή την ανεξέλεγκτη δημιουργία αντιγράφων μίας διεργασίας που απομυζεί τους πόρους του συστήματος θέτει περιορισμό στον αριθμό των διεργασιών που μπορεί να δημιουργήσει ένας χρήστης. Στο linux κάτι τέτοιο επιτυγχάνεται με την εντολή `ulimit -u 30`, περιορίζοντας των αριθμών τους στις τριάντα.

1.2 Δημιουργία αυθαίρετου δέντρου διεργασιών

Σε αυτό το ερώτημα ζητείται η γενίκευση του προηγούμενο προγράμματος στην δημιουργία ενός δέντρου διεργασιών, το οποίο αυτή τη φορά δίνεται ως είσοδος από αρχείο. Το πρόγραμμα βασίζεται σε αναδρομική συνάρτηση η οποία καλείται για κάθε κόμβο του δέντρου και δημιουργεί τις διεργασίες παιδιά του. Εάν ο κόμβος είναι φύλλο τότε εκτελεί την εντολή `sleep()` όπως και προηγουμένως ούτως ώστε να δοθεί χρόνος στον χρήστη να παρατηρήσει το δέντρο προτού αρχίσει να καταστρέφεται.

Η έξοδος του προγράμματος μας παρακάτω.

```
$ ./ask2-fork proc.tree
A
      B
            E
            F
      C
      D
A begins...
A forks B.
A forks C.
B forks E.
A forks D.
B forks F.
C is a leaf.
C sleeping...
E is a leaf.
E sleeping...
D is a leaf.
F is a leaf.
D sleeping...
F sleeping...


A(5671)─┬─B(5672)─┬─E(5674)
        │         └─F(5676)
        ├─C(5673)
        └─D(5675)


C exiting...
E exiting...
D exiting...
F exiting...
My PID = 5671: Child PID = 5673 terminated normally, exit status = 0
My PID = 5672: Child PID = 5674 terminated normally, exit status = 0
My PID = 5671: Child PID = 5675 terminated normally, exit status = 0
My PID = 5672: Child PID = 5676 terminated normally, exit status = 0
B exiting...
My PID = 5671: Child PID = 5672 terminated normally, exit status = 0
A exiting...
My PID = 5670: Child PID = 5671 terminated normally, exit status = 0
$
```

1) Όπως αναφέρεται στην εκφώνηση, όταν μία διεργασία εκκινεί, πράγμα που σηματοδοτείται με το μήνυμα begins, ξεκινάει να κατασκευάζει τις διεργασίες – παιδιά της και περιμένει μέχρις ότου να τερματίσουν ώστε να τερματίσει και η ίδια, τυπώνοντας αυτή τη φορά το μήνυμα exiting. Η σειρά με την οποία δημιουργούνται οι κόμβοι παραπάνω είναι A, B, C, E, D, F πράγμα που είναι τυχαίο καθώς υπάρχει racing μεταξύ των διεργασιών που τρέχουν παράλληλα και ο χρονοδρομολογητής είναι εκείνος που θα αποφασίσει σε ποια από τις ενεργές διεργασίες θα δώσει

προτεραιότητα. Τα μηνύματα τερματισμού απ' την άλλη εμφανίζονται με την ίδια σειρά που εμφανίζονται τα αντίστοιχα μηνύματα εκκίνησης για κάθε κόμβο. Αυτό συμβαίνει διότι πρώτα θα τερματίσει η διεργασία – φύλλο που κατασκευάστηκε πρώτη, αφότου εκτελέσει την εντολή αναστολής `sleep()`, ύστερα εκείνη που κατασκευάστηκε δεύτερη κοκ. Ωστόσο, εάν τρέξουμε πολλές φορές το πρόγραμμα θα παρατηρήσουμε ότι η σειρά εμφάνισης των μηνυμάτων begins διαφέρει κάθε φορά, κάτι που είναι αναμενόμενο καθώς δεν χρησιμοποιούμε κάποιο μηχανισμό συγχρονισμού των διεργασιών όπως είναι τα σήματα με αποτέλεσμα ο χρονοδρομολογητής να δίνει προτεραιότητα σε διαφορετικές διεργασίες κάθε φορά.

1.3 Αποστολή και χειρισμός σημάτων

Σε αυτό το ερώτημα ζητείται να μεταρρυθμίσουμε το προηγούμενο πρόγραμμα ώστε κάθε κόμβος, αφού δημιουργήσει όλες τις διεργασίες – παιδία του, να αναστέλλει τη λειτουργία του έως ότου να λάβει κατάλληλο σήμα εκκίνησης. Τότε, η διεργασία να «ξυπνάει» και να ενεργοποιεί διαδοχικά και όλα τα παιδιά της. Η αναστολή μίας διεργασίας γίνεται με την κλήση της `raise(SIGSTOP)` ενώ με τη χρήση της εντολής `kill(pid, SIGCONT)` μία διεργασία μητέρα ενημερώνει με σήμα εκκίνησης τα παιδιά της.

```
$ ./ask2-signals proc.tree
PID = 5779, name A, starting...
PID = 5779, A forks B.
PID = 5779, A forks C.
PID = 5780, name B, starting...
PID = 5779, A forks D.
PID = 5780, B forks E.
PID = 5781, name C, starting...
PID = 5781, C is a leaf, suspends.
PID = 5779, A suspends.
My PID = 5779: Child PID = 5781 has been stopped by a signal, signo = 19
PID = 5780, B forks F.
PID = 5782, name D, starting...
PID = 5782, D is a leaf, suspends.
PID = 5783, name E, starting...
PID = 5783, E is a leaf, suspends.
My PID = 5779: Child PID = 5782 has been stopped by a signal, signo = 19
PID = 5780, B suspends.
My PID = 5780: Child PID = 5783 has been stopped by a signal, signo = 19
PID = 5784, name F, starting...
PID = 5784, F is a leaf, suspends.
My PID = 5780: Child PID = 5784 has been stopped by a signal, signo = 19
My PID = 5779: Child PID = 5780 has been stopped by a signal, signo = 19
My PID = 5778: Child PID = 5779 has been stopped by a signal, signo = 19


A(5779)─┬─B(5780)─┬─E(5783)
        │         └─F(5784)
        ├─C(5781)
        └─D(5782)


PID = 5779, name = A is awake
PID = 5780, name = B is awake
PID = 5783, name = E is awake
PID = 5783, E exiting...
My PID = 5780: Child PID = 5783 terminated normally, exit status = 0
PID = 5784, name = F is awake
PID = 5784, F exiting...
My PID = 5780: Child PID = 5784 terminated normally, exit status = 0
PID = 5780, B exiting...
My PID = 5779: Child PID = 5780 terminated normally, exit status = 0
PID = 5781, name = C is awake
PID = 5781, C exiting...
My PID = 5779: Child PID = 5781 terminated normally, exit status = 0
PID = 5782, name = D is awake
PID = 5782, D exiting...
My PID = 5779: Child PID = 5782 terminated normally, exit status = 0
PID = 5779, A exiting...
My PID = 5778: Child PID = 5779 terminated normally, exit status = 0
$
```

1) Χρησιμοποιώντας σήματα αυτή τη φορά για να ελέγξουμε τη ροή του προγράμματος, καταφέραμε μία διεργασία να ξεκινάει ή να σταματά ανάλογα με το αν έχει δεχτεί μήνυμα εκκίνησης από τη διεργασία γονέα ή εάν έχει αναστείλει τον εαυτό της αντίστοιχα. Έτσι, δεν χρειάζεται τα φύλλα του δέντρου να εκτελέσουν τη `sleep()` για κάποιο χρονικό διάστημα ώστε να προλάβει ο χρήστης να παρατηρήσει το δέντρο. Επιπλέον, παρακάμπτουμε το πρόβλημα που δημιουργείται όταν έχουμε μονοπάτια με πάρα πολλούς κόμβους και δεν γνωρίζουμε αν ο χρόνος που έχουμε προστάξει τα φύλλα να κοιμηθούν είναι αρκετός ώστε να κατασκευαστεί όλο το δέντρο. Συνεπώς, η χρήση σημάτων εξασφαλίζει επικοινωνία και συγχρονισμό των διεργασιών ενώ στις περισσότερες περιπτώσεις το αποτέλεσμα του προγράμματος μας δίνεται συντομότερα λόγω της αποφυγής της εντολής `sleep()`.
2) Η `wait_for_ready_children()` για μία διεργασία γονέα περιμένει μέχρι όλα τα παιδία της να ανασταλούν μέσω της κλήσης `raise(SIGSTOP)`. Μόλις γίνει αυτό στέλνεται σε αυτά σήμα εκκίνησης μέσω της `kill(pid, SIGCONT)` και περιμένει μέχρι να τερματίσουν. Εάν παραλείπαμε αυτή τη συνάρτηση η σειρά ολοκλήρωσης των διεργασιών θα ήταν απρόβλεπτη και θα εξαρτιόταν από τον χρονοδρομολογητή του συστήματος. Με τη χρήση της εξασφαλίζουμε ότι όλα τα παιδιά μίας διεργασίας θα έχουν προλάβει να κάνουν `raise(SIGSTOP)` όταν ο γονέας τους στείλει σήμα επανεκκίνησης. Αν συμβεί συμβεί αυτό, το SIGCONT μπορεί να αγνοηθεί και στη συνέχεια το παιδί να μπει σε αναμονή από την οποία δε θα βγει ποτέ.

1.4 Παράλληλος υπολογισμός αριθμητικής έκφρασης

Χρησιμοποιώντας τις σωληνώσεις του UNIX, κατασκευάσαμε ένα πρόγραμμα που αποτιμάει αριθμητικές εκφράσεις που είναι οργανωμένες σε expression trees όπως αυτό που δίνεται στην εκφώνηση. Το πρόγραμμα, όπως και τα προηγούμενα, δέχεται ως είσοδο ένα αρχείο που περιέχει τους κόμβους ενός δέντρου σε DFS μορφή. Παρακάτω φαίνονται τα ενδιάμεσα μηνύματα που τυπώνουν οι διεργασίες και το τελικό αποτέλεσμα.

```
$ ./ask3-expr-tree expr.tree
+
      10
      *
            +
                  5
                  7
            4
PID: 6760, +: pipe created on process
PID: 6760, +: pipe created on process
PID: 6761, 10: this node is a leaf, writing on fd.
PID: 6760, +: child with value 10 came back
PID: 6762, *: pipe created on process
PID: 6762, *: pipe created on process
PID: 6763, +: pipe created on process
PID: 6764, 4: this node is a leaf, writing on fd.
PID: 6763, +: pipe created on process
PID: 6765, 5: this node is a leaf, writing on fd.
PID: 6763, +: child with value 5 came back
PID: 6766, 7: this node is a leaf, writing on fd.
PID: 6763, +: child with value 7 came back
PID: 6763, +: values returned from children's pipes are 5 and 7.
PID: 6762, *: child with value 12 came back
PID: 6762, *: child with value 4 came back
PID: 6762, *: values returned from children's pipes are 12 and 4.
PID: 6760, +: child with value 48 came back
PID: 6760, +: values returned from children's pipes are 10 and 48.
My PID = 6759: Child PID = 6760 terminated normally, exit status = 0


PID: 6759, main: result of the expression is 58

$
```

1) Καθώς γνωρίζουμε ότι κάθε ενδιάμεσος κόμβος (τελεστής) πρέπει να έχει ακριβώς δύο παιδιά (τελεσταίους), επιλέξαμε για κάθε έναν από αυτούς να κατασκευάσουμε δύο σωληνώσεις ώστε να μπορεί να επικοινωνεί ξεχωριστά με κάθε παιδί. Στην περίπτωση που βρισκόμαστε σε κάποιο φύλλο, το οποίο περιέχει έναν ακέραιο αριθμό, χρειαζόμαστε μονάχα ένα pipe που επικοινωνεί με τον γονέα και του μεταβιβάζει το περιεχόμενο του. Ακολουθήσαμε αυτή την υλοποίηση για λόγους πληρότητας καθώς εάν θέλουμε το δέντρο να περιλαμβάνει πράξεις για τις οποίες δεν ισχύει η αντιμεταθετική ιδιότητα, όπως αφαίρεση ή διαίρεση, είναι σημαντικό να γνωρίζουμε τη σειρά των τελεσταίων για να πάρουμε σωστό αποτέλεσμα. Στο συγκεκριμένο πρόγραμμα δεν υποστηρίζουμε αυτές τις πράξεις πράγμα που σημαίνει ότι δεν μας ενδιαφέρει η σειρά με την οποία θα εμφανιστούν στις σωληνώσεις τα αποτελέσματα των παιδιών και επομένως μπορεί να γίνει υλοποίηση και μόνο με ένα pipe για κάθε μη τερματικό κόμβο.
2) Σε ένα σύστημα όπου μπορούν να εκτελούνται παραπάνω από μία διεργασίες παράλληλα, μπορούμε να επιταχύνουμε σε μεγάλο βαθμό τη διαδικασία της αποτίμησης μίας τέτοιας έκφρασης μοιράζοντας τις επιμέρους πράξεις του δέντρου σε ξεχωριστές διεργασίες έναντι της ανάθεσης όλης της διαδικασίας σε μία μόνο διεργασία. Βέβαια, το πραγματικό σενάριο στο οποίο όντως έχουμε χρονικό κέρδος είναι όταν το expression tree είναι ισορροπημένο ώστε οι διεργασίες – πράξεις να ολοκληρώνονται ανά επίπεδο καθώς δεν θέλουμε μία μη τερματική διεργασία να περιμένει λίγο χρόνο για να επιστρέψει ένα παιδί και πολύ χρόνο για να επιστρέψει ένα άλλο.

Πηγαίος Κώδικας

Μαζί με την παρούσα αναφορά επισυνάπτουμε και τον φάκελο `forktree` στον οποίο συμπεριλαμβάνονται:

- Ένα τροποποιημένο `Makefile`, το οποίο πέραν από τα εκτελέσιμα της εκφώνησης, κατασκευάζει και τα εκτελέσιμα αρχεία του κώδικα που γράψαμε.
- `ask1-given-tree.c` το source file του ερωτήματος 1.1
- `ask2-fork.c` το source file του ερωτήματος 1.2
- `ask2-signals.c` το source file του ερωτήματος 1.3
- `ask3-expr-tree.c` το source file του ερωτήματος 1.4
- Τα headers και source files που μας δίνονταν από την εκφώνηση και είναι απαραίτητα για τη μεταγλώττιση των αρχείων μας.