Σχολή
Ηλεκτρολόγων Μηχανικών
και
Μηχανικών Υπολογιστών

Λειτουργικά Συστήματα

2η σειρά ασκήσεων

Ομάδα oslabb12
Γκούμας Βασίλης — 03113031
Ζαρίφης Νικόλαος — 03112178

Άσκηση 1

Ο πηγαίος κώδικας βρίσκεται στο αρχείο ask2-fork.c και παρακάτω βλέπουμε το output της εκτέλεσης του. Όπως θα δούμε και στα θεωρητικά ερωτήματα, η παραπάνω έξοδος παράχθηκε με show_pstree(getpid()).

Στη περίπτωση που είχαμε show_pstree(pid) θα έιχαμε το ίδιο αποτέλεσμα στο δέντρο, χωρίς όμως την γραμμή που περιέχει τα sh, pstree

```
oslabb12@leykada:~/ex2$ ./ask2-fork
A: Sleeping...
B: Sleeping...
C: Sleeping...
D: Sleeping...


ask2-fork(10996)─┬─A(10997)─┬─B(10998)──D(11000)
                 |          └─C(10999)
                 └─sh(11001)──pstree(11002)


C: Exiting...
D: Exiting...
My PID = 10997: Child PID = 10999 terminated normally, exit status = 17
My PID = 10998: Child PID = 11000 terminated normally, exit status = 13
B: Exiting...
My PID = 10997: Child PID = 10998 terminated normally, exit status = 19
A: Exiting...
My PID = 10996: Child PID = 10997 terminated normally, exit status = 16
```

1ο ερώτημα

Άμα τερματίσουμε πρόωρα μια διεργασία, τότε όλα τα παιδιά θα περάσουν σε μια κατάσταση που ονομάζεται zombie καθώς δεν θα υπάρχει ο γονιός για να τα κάνει wait. Σε αυτή την περίπτωση , η πρακτική που ακολουθείται στα $unix$ λειτουργικά συστήματα είναι τα παιδιά zombies να γίνονται παιδιά μιας διεργασίας με $pid = 1$, η οποία κάνει διαρκώς wait.

2ο ερώτημα

Γράφοντας show_ps_tree(getpid()) αντί για show_ps_tree(pid), θα εμφανιστεί το δέντρο συμπεριλαμβάνοντας την διεργασία του προγράμματός μας (ask2-fork), η οποία κάνει fork μια διεργασία shell η οποία με τη σειρά της καλεί την pstree που εκτυπώνει το ζητούμενο δέντρο.

3ο ερώτημα

Αν ο διαχειριστής δεν έχει θέσει όριο στον αριθμό διεργασιών που μπορεί να δημιουργήσει ένας χρήστης τότε είναι πιθανό λόγω κακής/κακόβουλης χρήσης να εξαντληθούν οι πόροι του μηχανήματος, καθιστώντας το αχρησιμοποιήτο απο τους υπόλοιπους χρήστες. Χαρακτηριστικό παράδειγμα κακόβουλης χρήσης είναι το forkbomb που είναι μια συνάρτηση που συνεχώς κάνει fork τον εαυτό της.

Άσκηση 2

Ο κώδικας της άσκησης βρίσκεται στο αρχείο ask2-tree.c. Επίσης περιέχονται μερικά testcases που τρέξαμε με όνομα *.in και *.out αντίστοιχα καθώς και το output της εκτέλεσής του.

```
child with 11005 created
child with 11006 created
child with 11007 created
child with 11008 created
child with 11009 created


A(11004)─┬─B(11005)─┬─E(11008)
         |          └─F(11009)
         ├─C(11006)
         └─D(11007)


My PID = 11004: Child PID = 11006 terminated normally, exit status = 1
My PID = 11004: Child PID = 11007 terminated normally, exit status = 1
My PID = 11005: Child PID = 11008 terminated normally, exit status = 1
My PID = 11005: Child PID = 11009 terminated normally, exit status = 1
My PID = 11004: Child PID = 11005 terminated normally, exit status = 1
My PID = 11003: Child PID = 11004 terminated normally, exit status = 1
```

Ερώτημα 1ο

Όπως βλέπουμε τα μηνύματα έναρξης εμφανίζονται κατα βάθος bfs, καθώς κάθε διεργασία δημιουργεί με ένα for τα παιδιά της.Τα μηνύματα τερματισμού αντίστοιχα, δεν έχουν κάποιο μοτίβο, καθώς χωρίς επιπλέον έλεγχο η σειρά που θα εμφανιστούν εξαρτάται απο τη σειρά που θα δώσει ο scheduler στις διεργασίες.

Άσκηση 3

Ο κώδικας της άσκησης βρίσκεται στο αρχείο ask2-signals.c

```
oslabb12@leykada:~/ex2$ ./ask2-signals proc.tree
PID = 11014, name A, starting...
PID = 11015, name B, starting...
PID = 11016, name C, starting...
My PID = 11014: Child PID = 11016 has been stopped by a signal, signo = 19
PID = 11017, name D, starting...
My PID = 11014: Child PID = 11017 has been stopped by a signal, signo = 19
PID = 11018, name E, starting...
My PID = 11015: Child PID = 11018 has been stopped by a signal, signo = 19
PID = 11019, name F, starting...
My PID = 11015: Child PID = 11019 has been stopped by a signal, signo = 19
My PID = 11014: Child PID = 11015 has been stopped by a signal, signo = 19
My PID = 11013: Child PID = 11014 has been stopped by a signal, signo = 19


A(11014)─┬─B(11015)─┬─E(11018)
         |          └─F(11019)
         ├─C(11016)
         └─D(11017)


PID = 11014, name = A is awake
PID = 11017, name = D is awake
I am D and I have finished
My PID = 11014: Child PID = 11017 terminated normally, exit status = 0
PID = 11016, name = C is awake
I am C and I have finished
My PID = 11014: Child PID = 11016 terminated normally, exit status = 0
PID = 11015, name = B is awake
PID = 11019, name = F is awake
I am F and I have finished
My PID = 11015: Child PID = 11019 terminated normally, exit status = 0
PID = 11018, name = E is awake
I am E and I have finished
My PID = 11015: Child PID = 11018 terminated normally, exit status = 0
I am B and I have finished
My PID = 11014: Child PID = 11015 terminated normally, exit status = 0
I am A and I have finished
My PID = 11013: Child PID = 11014 terminated normally, exit status = 0
```

1ο ερώτημα

Στις ασκήσεις 1 και 2 που έγινε χρήση της sleep(), χρειάστηκε να περιμένουμε ένα χρονικό διάστημα που εμείς ορίζουμε εκ των προτέρων, το οποίο υπολογίζουμε ότι θα είναι αρκετό για να δημιουργηθεί ολόκληρο το δέντρο διεργασιών, πριν το τυπώσουμε.

Προφανώς αυτή η διαδικασία δεν είναι αρκετή για σε περίπτωση που το δέντρο διεργασιών γίνεται μεγάλο καθώς δεν κάνει scale και απαιτεί περιττή δουλειά απο τον προγραμματιστή. και γενικά όταν θέλουμε να κάνουμε ακριβή συγχρονισμό διεργασιών.

Χρησιμοποιώντας σήματα έχουμε μεγαλύτερο έλεγχο στον συγχρονισμό καθώς ξυπνάμε και κοιμίζουμε τις διεργασίες ακριβώς τη στιγμή που τις χρειαζόμαστε, έχοντας ταυτόχρονα πετύχει ασύγχρονο τρόπο επικοινωνίας μεταξύ των δεργασιών.

2ο ερώτημα

Όπως βλέπουμε και απο την υλοποίηση της, η wait_for_ready_children() σιγουρεύεται πως όλα τα παιδιά της εκάστοτε διεργασίας που την κάλεσε, έχουν γίνει pause και δεν έχουν τερματίσει για κάποιον άλλο λόγο.

Αυτό το καταφέρνει εξετάζοντας τη κατάσταση τερματισμού μέσω των τιμών που επιτρέφει η waitpid() και συγκεκριμένα μέσω του WIFSTOPPED(status).

Στο πρόγραμμά μας, η wait_for_ready_children καλείται αναδρομικά για κάθε παιδί και στη συνέχεια κάνουν pause τον εαυτό τους. Επομένως η κλήση της wait_for_ready_children που βρίσκεται στη main, περιμένει να σχηματιστεί ολόκληρο το δέντρο πριν καλεστεί η συνάρτηση που θα το τυπώσει.

Η χρήση της εξασφαλίζει ότι γνωρίζουμε την κατάσταση του δέντρου πριν προσπαθήσουμε να ξυπνήσουμε παιδιά τα οποία ενδεχομένως να μην έχουν γίνει ακόμα pause ή να μην έχουν καν δημιουργηθεί.

Άσκηση 4

Ο κώδικας της άσκησης βρίσκεται στο ask2-pipes.c Παρακάτω φαίνεται το αποτέλεσμα του προγράμματός μας για το testcase που μας δώθηκε.

```
oslabb12@leykada:~/ex2$ ./ask2-pipes expr.tree
I am 10 and I have write : 10
My PID = 11353: Child PID = 11354 terminated normally, exit status = 0
I am + and I have read : 10
I am 4 and I have write : 4
My PID = 11355: Child PID = 11357 terminated normally, exit status = 0
I am * and I have read : 4
I am 5 and I have write : 5
My PID = 11356: Child PID = 11358 terminated normally, exit status = 0
I am + and I have read : 5
I am 7 and I have write : 7
My PID = 11356: Child PID = 11359 terminated normally, exit status = 0
I am + and I have read : 7
I am + and I have write : 12
My PID = 11355: Child PID = 11356 terminated normally, exit status = 0
I am * and I have read : 12
I am * and I have write : 48
My PID = 11353: Child PID = 11355 terminated normally, exit status = 0
I am + and I have read : 48
I am + and I have write : 58
My PID = 11352: Child PID = 11353 terminated normally, exit status = 0
I have read : 58
58
```

1ο ερώτημα

Στο πρόγραμμα μας κάθε διεργασία αλληλεπιδρά με δύο σωληνώσεις, με αυτή που γράφει και με αυτή που διαβάζει. Εξαίρεση βέβαια αποτελούν οι

ακραίες διεργασίες, τα φύλλα μόνο γράφουν το αποτέλεσμα τους ώστε να το μεταβιβάσουν σε μεγλύτερο επίπεδο, ενώ η ρίζα απλά διαβάζει το αποτέλεσμα απο τις γονικές της διεργασίες και το τυπώνει.

Για την επικοινωνία της γονικής διεργασίας με τα παιδιά της, χρησιμοποιήθηκε ένα pipe, το οποίο μοιράζονται τα δύο παιδιά.Στη συγκεκριμένη περίπτωση το κοινό pipe για τα παιδιά είναι υλοποιήσιμο καθώς δεν χρειάζετε να ξέρουμε ποιο αποτέλεσμα προήλθε απο ποιο παιδί, καθώς η πράξη της πρόσθεσης και του πολλαπλασιασμού είναι αντιμεταθετικές.

Επομένως στη γενική περίπτωση, όπως πχ στην πράξη της αφαίρεσης, ή οποιασδήποτε λειτουργία απαιτεί να ξέρουμε απο ποιο παιδί ήρθε κάποιο αποτέλεσμα, θα αναγκαστούμε να χρησιμοποήσουμε ένα pipe ανα παιδί, είτε να παρακαλοθούμε πότε τερματίζει κάθε παιδί.

2ο ερώτημα

Σε ένα πολυεπεξεργαστικό σύστημα, οι διαφορετικές διεργασίες/thread που δημιουργούμε για τους κόμβους, είναι δυνατόν να εκτελούνται σε διαφορετικές cpu, με αποτέλεσμα να επιτυγχάνεται παραλληλοποίηση και να έχουμε performance boost και αρκετά γρηγορότερο χρόνο εκτέλεσης.

Για να έχουμε όσο δυνατόν μεγαλύτερο όφελος απο την παραλληλοποίηση, θα πρέπει το πρόγραμμα μας να μπορεί να διαχωριστεί σε τμήματα με χαμηλά ή μηδαμινά dependencies.

Για το παράδειγμα μας, μια περίπτωση εκφυλισμού όπου η παράλληλη εκτέλεση είναι ισοδύναμη με την σειριακή είναι όταν επιτρέπεται η χρήση παρενθέσεων. Στη συγκεκριμένη περίπτωση, μπορεί ένα απο καιρό έτοιμο αποτέλεσμα να stallάρει καθώς αναμένει αποτέλεσμα απο τις nested παρενθέσεις. Το δέντρο της διεργασίας στην παραπάνω περίπτωση θα ήταν εντελώς imbalanced προς τα δεξιά ή αριστερά. Αντίθετα, όσο πιο κοντά είμαστε σε πλήρες δέντρο, τόσο μειώνονται τα dependencies και έχουμε μεγαλύτερο όφελος απο την παραλληλία.

Προαιρετικές

1ο ερώτημα

Όπως είδαμε και στο 2ο ερώτημα της 4ης άσκησης, ο χρόνος υπολογισμού μιας αριθμητικής έκφρασης εξαρτάται απο τη μορφή του δέντρου.Όσο πιο

κοντά σε πλήρες είναι το δέντρο μας, τόσο μεγαλύτερο speedup έχουμε απο την παραλληλοποιήση και χρόνος εκτέλεσης προσεγγίζει το O(logn), όπου n το πλήθος των κόμβων του δέντρου.Αν το δέντρο εκφυλίζεται προς τα αριστερά/δεξιά τότε ο χρόνος εκτέλεσης τείνει να γίνει O(n) .

Άμα έχουμε ένα δέντρο με n κόμβους, το οποίο είναι κατα προσέγγιση πλήρες, τότε το μέγιστο πλήθος ταυτόχρονων διεργασιών που θα τρέξουν είναι n επομένως δεν έχει νόημα να προσθέσουμε περισσότερους επεξεργαστές. Επομένως χρησιμοποιώντας n επεξεργαστές μπορούμε να πετύχουμε ιδανικό speedup χωρίς να μένει ανεξυπηρέτητος κάποιος κόμβος.Περισσότεροι επεξεργαστές δεν θα αξιοποιηθούν και επομένως αποτελούν σπατάλη πόρων.Λιγότεροι ίσως βρεθούν σε κατάσταση που είναι όλοι απασχολημένοι και κάποια διεργασία περιμένει να τρέξει.

2ο ερώτημα

Το πλεονέκτημα της υβριδικής υλοποίησης είναι ότι δε δεσμεύει επεξεργαστές που υπάρχει περίπτωση να βρίσκονται σε κατάσταση αναμονής, αν υπάρχουν μεγάλα dependencies.

Αυξάνοντας το μ έχουμε βελτίωση απόδοσης, με την προυπόθεση ότι στο νέο βάθος το δέντρο μας συνεχίζει να είναι κατα προσέγγιση πλήρες.

Με μικρότερο μ απο την άλλη πλευρά είναι πιθανό να έχουμε μειωμένο χρόνο εκτέλεσης καθώς δεν αξιοποιούμε την παραλληλία της έκφρασης.

Το βάθος ενός πλήρους δυαδικού δέντρου με n κόμβους είναι log(n). Επομένως μέχρι αυτό το επίπεδο περιμένουμε να έχουμε ένα σχετικά μη εκφυλλισμένο δέντρο. Σε μεγαλύτερο βάθος περιμένουμε να έχουμε αυξημένο nesting και επομένως δεν υπάρχει όφελος στην παραλληλία.