Αλεξίου Περικλής 03122079
Τσιμπλάκη Πηνελόπη Άννα 03122097

1η Άσκηση Λειτουργικών Συστημάτων
Κλήσεις συστήματος, διεργασίες και διαδιεργασιακή επικοινωνία

Πρόλογος

Στην παρούσα αναφορά παρουσιάζεται το σκεπτικό πίσω από κάθε πρόγραμμα που γράψαμε στο πλαίσιο της άσκησης 1. Επίσης φαίνονται σε στιγμιότυπα εκτελέσεις κάθε προγράμματος που υποδεικνύουν την λειτουργικότητα του. Δεν παρατίθενται ολόκληροι οι κώδικες, καθώς αυτοί συμπεριλαμβάνονται σε κοινό αρχείο με την αναφορά και μπορούν να βρεθούν εκεί. Αυτό για να διευκολυνθεί η ανάγνωσή της.

<u>Μέρος 1</u>: Ανάγνωση και εγγραφή αρχείων στη C και με τη βοήθεια κλήσεων συστήματος

Στο αρχείο a1_1.c, υλοποιήσαμε main function που αναζητά πόσες φορές εμφανίζεται ένας χαρακτήρας μέσα σε ένα αρχείο εισόδου και γράφει το αποτέλεσμα σε ένα αρχείο εξόδου. Τα αρχεία εισόδου και εξόδου, καθώς και ο χαρακτήρας προς αναζήτηση, δίνονται από τον χρήστη στη γραμμή εντολών.

Το πρόγραμμα έχει απλή λειτουργία. Αρχικά ανοίγει τα αρχεία εγγραφής και ανάγνωσης με την open() και τα κατάλληλα flags και modes. Αποθηκεύει τα file descriptors τους στις μεταβλητές τύπου integer fdw και fdr, αντίστοιχα.
Έπειτα, με την read() διαβάζει επαναληπτικά το αρχείο fdr (1024 bytes τη φορά που είναι το μέγεθος του buffer). Μετά από κάθε επανάληψη, συγκρίνει κάθε χαρακτήρα που αποθηκεύτηκε στον buffer με τον χαρακτήρα αναζήτησης, ενημερώνοντας, αν χρειάζεται, τον συνολικό αριθμό εμφανίσεων στον μετρητή count. Η διαδικασία τερματίζεται με την ανάγνωση του χαρακτήρα EOF. Αυτό ελέγχεται μέσω της read() που θα επιστρέψει 0. Τέλος, με την sprintf() διαμορφώνουμε ένα buffer με το κείμενο που θέλουμε να γράψουμε μαζί με το αποτέλεσμα μας και με την write() το γράφουμε στο αρχείο εγγραφής, fdw. Μόλις τελειώσουμε με τα αρχεία ανάγνωσης και εγγραφής, τα κάνουμε close() με τον αντίστοιχο file descriptor.
Μετά από κάθε system call, φροντίζουμε ώστε, σε περίπτωση που επιστραφεί τιμή ενδεικτική προβληματικής εκτέλεσης, να εκτυπωθεί στο standard error μήνυμα κατατοπιστικό για την προέλευση του σφάλματος.

Ακολουθεί πίνακας που συνδέει τα flags και τα modes που χρησιμοποιήθηκαν με τις λειτουργίες που επιτελούν.

<u>fdr flags</u>:

O_RDONLY	Άνοιγμα μόνο για ανάγνωση

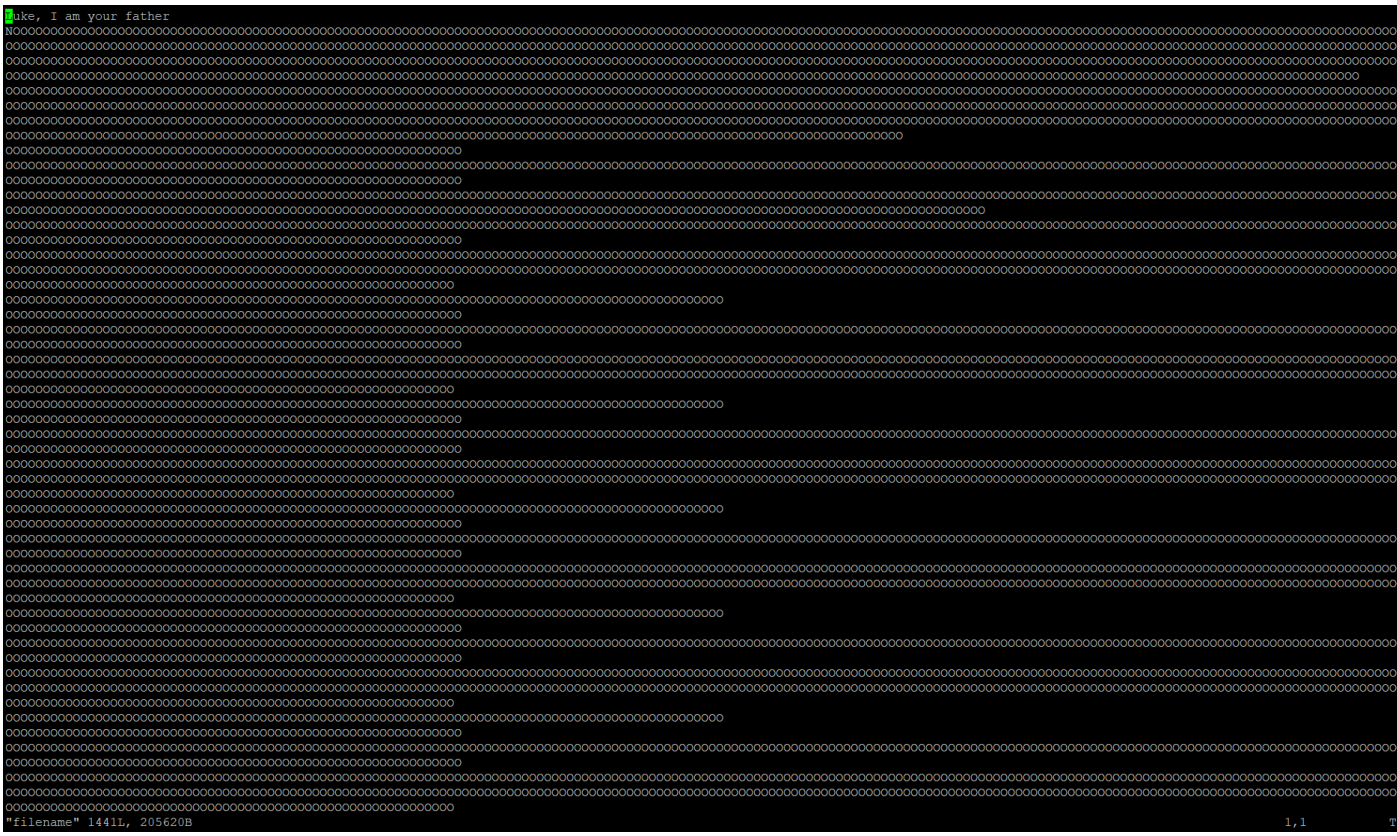
<u>fdw flags</u>:

O_WRONLY	Άνοιγμα μόνο για εγγραφή
O_CREAT	Εάν το αρχείο δεν υπάρχει, να δημιουργηθεί
O_TRUNC	Αν το αρχείο υπάρχει, το μήκος του να γίνει 0

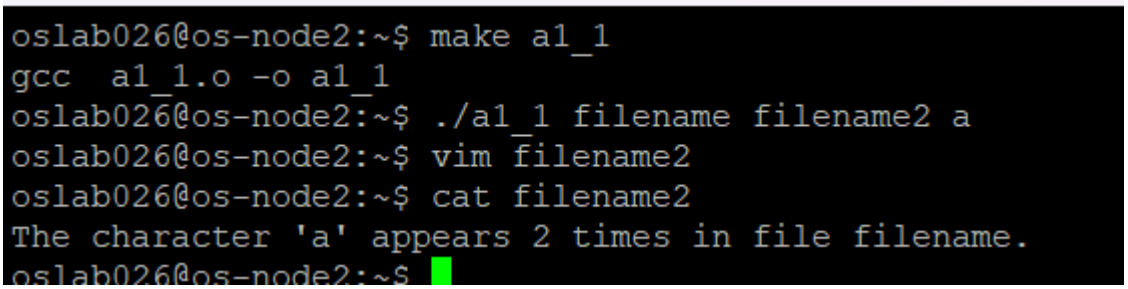
<u>fdw mode:</u>

S_IRUSR	Δίνει στον ιδιοκτήτη άδεια ανάγνωσης
S_IWUSR	Δίνει στον ιδιοκτήτη άδεια εγγραφής

Για να ελέγξουμε τη λειτουργία του προγράμματος, χρησιμοποιήσαμε ως αρχείο ανάγνωσης το filename. Περιείχε 2 εμφανίσεις του χαρακτήρα ‘a’ μεταξύ περισσότερων από 1024 χαρακτήρων.



Ως αρχείο εγγραφής χρησιμοποιήσαμε το filename2.



Το περιεχόμενο του filename2 μετά την εκτέλεση του a1_1 είναι το επιθυμητό.

Μέρος 2: Δημιουργία Διεργασιών

a1_2 (1)

Το πρόγραμμα a1_2, δημιουργεί κατά την εκτέλεση του, μια διεργασία-παιδί αντίγραφο του, με την κλήση συστήματος fork(). Η ροή του προγράμματος διακλαδώνεται, εκτελώντας διαφορετική λειτουργία ανάλογα με το αν βρισκόμαστε στη διεργασία γονιό ή στη διεργασία παιδί. Διαχωρίζουμε τις διεργασίες με βάση την τιμή που επιστρέφει η fork, η οποία αποθηκεύεται στη μεταβλητή p, τύπου pid_t. Στη διεργασία παιδί, ισχύει p=0, ενώ, στη διεργασία γονέα, η p γίνεται ίση με το Process ID του παιδιού της. Οι διεργασίες καλούν τη συνάρτηση child() και parent() αντίστοιχα, πριν τον τερματισμό τους. Εάν επιστραφεί p<0, τυπώνουμε στο stderr ότι προέκυψε σφάλμα κατά την εκτέλεση της fork().

Child():

Ανακτά το PID της διεργασίας από την οποία καλείται και του γονιού αυτής, με τις συναρτήσεις getpid() και getppid() αντίστοιχα. Έπειτα, γράφει τις πληροφορίες που συνέλεξε στο standard output.

Parent():

Περιμένει το θάνατο του παιδιού της, με την κλήση συστήματος wait(), ώστε να την απελευθερώσει από την κατάσταση “zombie”. Η wait() δέχεται σαν παράμετρο τη διεύθυνση της μεταβλητής status, στην οποία θα γράψει το exit status του παιδιού. Η συνάρτηση wait επιστρέφει το PID του νεκρού παιδιού. Αφού το συλλέξει, η parent() το αναφέρει στο standard output.

Παρατηρούμε ότι το πρόγραμμα λειτουργεί όπως αναμένεται.

```
oslab026@os-node2:~$ make a1_2
gcc  a1_2.o -o a1_2
oslab026@os-node2:~$ ./a1_2
Hello world! I am process 1173381, child of 1173380.
My late child was called 1173381.
```

a1_2 (2)

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <sys/wait.h>
#include <stdlib.h>

int main() {
  pid_t p, mypid=-1;
  char *x = "parent\n";
  p=fork();


  if(p<0){
      perror("fork");
      exit(1);
  } else if(p==0) {
      x = "child\n";
      write(1,x,strlen(x));
  } else {
      int status;
      wait(&status);
      write(1,x,strlen(x));
  }

  return 0;
}
```

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <sys/wait.h>
#include <stdlib.h>

int main() {
  pid_t p, mypid=-1;
  char *x = "parent\n";
  p=fork();


  if(p<0){
      perror("fork");
      exit(1);
  } else if(p==0) {
      x = "child\n";
      write(1,x,strlen(x));
  } else {
      x = "parent-new\n";
      int status;
      wait(&status);
      write(1,x,strlen(x));
  }

  return 0;
}
```

Στο πρόγραμμα a1_22 ορίζουμε μία μεταβλητή x, char pointer, προτού γίνει fork(). Συγκεκριμενα την αρχικοποιουμε ως x = “parent”. Διαχωρίζουμε τη ροή του προγράμματος, ανάλογα με το αν βρισκόμαστε στη συνάρτηση παιδί ή τη συνάρτηση γονέα, με βάση την τιμή επιστροφής της fork(). Ανατεθετουμε νέα τιμή στη x στο παιδί, x=”child”. Βλέπουμε πως η νέα ανάθεση στο παιδί δεν επηρεάζει την τιμή που έχει ο γονέας. Για λόγους αποδοτικότητας, η fork() επιτρέπει στην συνάρτηση child() να διαβάζει από τον χώρο μνήμης του γονιού της, ώσπου να “θελήσει” να αλλάξει το περιεχόμενο της. Σε αυτήν την περίπτωση, το λειτουργικό σύστημα βάζει τον πίνακα σελίδων της διεργασίας παιδί να δείχνει σε μια διαφορετική φυσική σελίδα μνήμης, αρχικοποιημένη σε τιμές όμοιες του γονιού. Η μέθοδος αυτή λέγεται “copy on write”. Αλλαγές στο περιεχόμενο μεταβλητών μετά από τη διακλάδωση,αφορούν τιμές διαφορετικών θέσεων μνήμης.
Η δυϊκότητα της μεταβλητής x γίνεται αντιληπτή όταν οι συναρτήσεις γράφουν την τιμή x που αντιλαμβάνονται στο stdout.

```
oslab026@os-node2:~$ make a1_22
gcc    -c -o a1_22.o a1_22.c
gcc  a1_22.o -o a1_22
oslab026@os-node2:~$ ./a1_22
child
parent
```

Επαναλαμβάνουμε το παραπάνω πείραμα, μεταβάλλοντας μόνο στη διεργασία γονιό την τιμή του parent σε parent-new. Ο διαχωρισμός των μεταβλητών γίνεται ξανά αισθητός.

```
oslab026@os-node2:~$ ./a1_22
child
parent-new
```

a1_2 (3)

Για το ερώτημα 2.3, ο γονιός είναι υπεύθυνος για την διαχείριση του αρχείου ανάγνωσης καθώς και για την εγγραφή, ενώ το παιδί επωμίζεται μόνο την αρμοδιότητα να βρει το πλήθος των εμφανίσεων του χαρακτήρα.
Ο γονέας ανοίγει τα δύο αρχεία για εγγραφή και ανάγνωση και ορίζει την integer μεταβλητή count. Δημιουργεί έναν πίνακα ακεραίων pfd[2], στον οποίο θα αποθηκευτούν τα file descriptors των άκρων του σωλήνα διαδιεργασιακής επικοινωνίας, που δημιουργούμε με τη κλήση συστήματος pipe(pfd).
Μετά την κλήση συστήματος fork(), η ροή του προγράμματος διακλαδώνεται ανάλογα με το αν βρισκόμαστε στη διεργασία παιδί ή γονιό.

Παιδί:

Κλείνει το άκρο ανάγνωσης του pipe, εφόσον η κατεύθυνση του σωλήνα είναι από το παιδί προς τον γονιό.
Το παιδί καλεί μια συνάρτηση child() που πραγματοποιεί την αναζήτηση του χαρακτήρα στο αρχείο αναζήτησης. Η child() λαμβάνει ως παραμέτρους το file descriptor του αρχείου αναζήτησης και τον χαρακτήρα προς αναζήτηση. Επιστρέφει τον αριθμό εμφανίσεων του χαρακτήρα, ο οποίος αποθηκεύεται στο count.
Με την κλήση συστήματος write(), η οποία λαμβάνει ως παράμετρο τη διεύθυνση του count και το μέγεθος του, γράφει στο άκρο εγγραφής του pipe το περιεχόμενο του.

Γονιός:

Κλείνει το άκρο εγγραφής του pipe, εφόσον η κατεύθυνση του σωλήνα είναι από το παιδί προς τον γονιό.
Ο γονιός μπλοκάρει στη wait(), ώσπου να συλλέξει το exit status του νεκρού παιδιού του. Διαβάζει το περιεχόμενο του pipe και το αποθηκεύει στο δικό του instance του count.
΄Επειτα καλεί την συνάρτηση parent() η οποία γράφει στο αρχείο εγγραφής πόσες φορές συναντήσαμε τον χαρακτήρα αναζήτησης στο αρχείο ανάγνωσης.
Πριν τον τερματισμό τους, οι διεργασίες κλείνουν όλα τα ανοιχτά file descriptors.

Το πρόγραμμα λειτουργεί ικανοποιητικά, όπως φαίνεται παρακάτω:

```
oslab026@os-node1:~$ cat book2
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
oslab026@os-node1:~$ ./a1_23 book2 book u
oslab026@os-node1:~$ cat book
The character 'u' appears 120 times in file book2.
```

a1_2 (4)

Στο ερώτημα 1.2.4 ζητείται να δημιουργήσουμε μια διεργασία που εκτελεί τον κώδικα του αρχείου a1_exp.c, ο οποίος δόθηκε με την εκφώνηση της άσκησης.

Λειτουργία προγράμματος:

Μετά την κλήση της fork(), στη διεργασία παιδί καλείται η execv(). Η execv() αντικαθιστά τον χώρο διευθύνσεων που βλέπει η διεργασία παιδί, ο οποίος αρχικά ταυτίζεται με αυτόν του γονιού, με του εκτελέσιμου αρχείου που λαμβάνει ως πρώτη παράμετρο. Ως δεύτερη παράμετρο, δέχεται έναν πίνακα char pointer. Πρώτο του στοιχείο είναι το όνομα του εκτελέσιμου αρχείου του οποίου τον κώδικα θα τρέξει η διεργασία-παιδί. Ακολουθείται από τις παραμέτρους εκτέλεσης του και τερματίζει με τον χαρακτήρα '\0'.
Το πρόγραμμα λειτουργεί όπως αναμένεται.

```
oslab026@os-node1:~$ cat filey
aaaaaa
oslab026@os-node1:~$ ./a1_24 filey book u
oslab026@os-node1:~$ cat book
The character 'u' appears 0 times in file filey.
oslab026@os-node1:~$ ./a1_24 filey book a
oslab026@os-node1:~$ cat book
The character 'a' appears 6 times in file filey.
```

Μέρος 3ο: Διαδιεργασιακή επικοινωνία

Στο 3ο μέρος της άσκησης, αναζητούμε τον αριθμό εμφανίσεων του χαρακτήρα αναζήτησης στο αρχείο ανάγνωσης ταυτόχρονα με P διεργασίες. Επίσης, όταν πληκτρολογείται Ctrl+C, δηλαδή όταν οι διεργασίες λαμβάνουν το σήμα SIGINT, στην περίπτωση μας από hardware interrupt, ο χρήστης να ενημερώνεται για τον αριθμό των ενεργών διεργασιών-παιδιών.
Ο κλήσεις συστήματος που αξιοποιούμε για να επιτύχουμε τα παραπάνω είναι οι:

1. **lseek():**
   Λαμβάνει ως παραμέτρους:
   - τον file descriptor του αρχείου του οποίου το offset θέλουμε να μετακινήσουμε
   - τον αριθμό offset bytes κατά τα οποία θέλουμε να μετακινήσουμε το offset, με τύπο δεδομένων off_t
   - έναν ακέραιο που αντιστοιχεί σε ένα από τα παρακάτω whence directives

`SEEK_SET`	Το offset ισούται με τα offset bytes
`SEEK_END`	Τα offset bytes προστίθενται στο μέγεθος του αρχείου

   Επιστρέφει το offset που αντιστοιχεί στον file descriptor που δόθηκε.

2. **signal():**
   Λαμβάνει ως παραμέτρους:
   - το σήμα (int) στο οποίο η διεργασία θέλουμε να ανταποκριθεί με τρόπο διαφορετικό από τον προκαθορισμένο
   - μια συνάρτηση void handler η οποία δέχεται ως παράμετρο τον ακέραιο αριθμό που αντιστοιχεί στο σήμα και εκτελείται αντί για την προκαθορισμένη ρουτίνα, σε περίπτωση που το σήμα ληφθεί

Λειτουργία προγράμματος:

Ο αριθμός διεργασιών P ορίζεται ως 6 με #define, ώστε να είναι απλή η διαδικασία μεταβολής του. Αρχικοποιείται σε 0 ακέραιος global μετρητής των νεκρών παιδιών, dead_proc. Οι διεργασίες γεννιούνται μέσα από ένα for το οποίο εκτελεί fork() P φορές και αντιστοιχεί κάθε νέα διεργασία σε index j 0 έως P-1.

Γονιός:

1. Αρχικά, ορίζουμε τον handler για το SIGINT με τη signal(). Λειτουργία του handler είναι να τυπώνει τον αριθμό ζωντανών παιδιών P-dead_proc στο standard output.
2. Ανοίγει το αρχείο εγγραφής και το αρχείο ανάγνωσης, με τα κατάλληλα flags.
3. Καλέι την lseek() με μηδενικό αριθμό offset byte και whence SEEK_END ώστε να υπολογίσει το μέγεθος του αρχείου ανάγνωσης. Κλείνει, έπειτα, το αρχείο ανάγνωσης (εναλλακτική υλοποίηση από την fstat όπως φαίνεται παρακάτω).
4. Δημιουργεί έναν πίνακα int pfd[P][2], ώστε να αποθηκεύσει τα άκρα εγγραφής και ανάγνωσης του pipe για κάθε παιδί. Με χρήση της fcntl() κάνει τα file descriptors ανάγνωσης non-blocking. Προτιμούμε να έχουμε ξεχωριστό pipe για κάθε παιδί, ώστε να μην υπάρχει διαπλοκή των δεδομένων και να μπορούμε να αναγνωρίζουμε την προέλευση των μηνυμάτων.
5. Μετά τα fork(), αρχικοποιεί τον μετρητή χαρακτήρων count σε 0. Κάνει polling ελέγχοντας διαδοχικά αν γράφτηκε κάτι στο pipe κάθε παιδιού. Αν το αποτέλεσμα του ελέγχου είναι θετικό, προσθέτει τον αριθμό μετρημένων χαρακτήρων subcount στο count και, αφού λάβει ένα pid νεκρού παιδιού η wait(), αυξάνει το dead_proc κατά 1.
   Σε αυτό το σημείο, αξίζει να σημειωθεί ότι η waitpid() ίσως να οδηγούσε σε καλύτερη οργάνωση του κώδικα, καθώς θα βεβαιωνόμασταν ότι δεν είναι zombie το παιδί του οποίου το αποτέλεσμα μέτρησης μόλις διαβάσαμε.
6. Γράφει τον συνολικό αριθμό εμφανίσεων του χαρακτήρα αναζήτησης που μετρήθηκαν στο αρχείο εγγραφής.

Παιδί:

1. Ορίζουμε SIG_IGN handler για το SIGINT με τη signal(), ώστε να μην τερματίζει με τη λήψη του.
2. Ανοίγει το αρχείο ανάγνωσης, ώστε να αποκτήσει δικό του file descriptor. Θα δημιουργούταν σύγχυση αν όλες οι διεργασίες-παιδιά μοιράζονταν ένα file descriptor.
3. Καλεί τη συνάρτηση child() και αποθηκεύει το αποτέλεσμα της στη μεταβλητή subcount, της οποίας το περιεχόμενο γράφει στο pipe.

   child():
   Επιστρέφει τον αριθμό χαρακτήρων αναζήτησης που βρήκε η διεργασία.
   Με την lseek(), το offset του file descriptor κάθε διεργασίας ορίζεται ως j*size/P. Το αρχείο ανάγνωσης χωρίζεται, δηλαδή, σε P τμήματα και κάθε παιδί αναλαμβάνει να αναζητήσει ένα από αυτά. Τα μήκη των τμημάτων που επεξεργάζονται οι διεργασίες 0 έως P-2 είναι size/P, ενώ η P-1 αναζητά

χαρακτήρες στο τμήμα με αρχή (P-1)*size/P και τέλος το τέλος του αρχείου ανάγνωσης.

Έλεγχος προγράμματος:

Το πρόγραμμα μετρά ορθά τους ζητούμενους χαρακτήρες, όπως φαίνεται στο παρακάτω στιγμιότυπο.

```
oslab026@os-node1:~$ ./temp book2 book u
oslab026@os-node1:~$ cat book
The character 'u' appears 120 times in file book2.
oslab026@os-node1:~$ cat book2
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
```

Προσθέτουμε την εντολή sleep(P-i), όπου i το αριθμός της διεργασίας στην child, ώστε, εισάγοντας τεχνητές καθυστερήσεις P-i second, να μπορούμε να ελέγξουμε πως συμπεριφέρεται το πρόγραμμα μετά τη λήψη του SIGINT. Φροντίζουμε, επίσης, ώστε κάθε διεργασία να εκτυπώνει το αναγνωριστικό της και τον αριθμό εμφανίσεων που μέτρησε. Με αυτόν τον τρόπο, ελέγχουμε πόσες διεργασίες έχουν ολοκληρώσει την αναζήτηση τους.

```c
int child(int fd, char c, int i, long size){
        sleep(P-i);
        long seg = size/P;
        lseek(fd, (off_t)(i*seg), SEEK_SET);
        char buff[10];
        ssize_t rcnt;
        int cnt = 0; int check = 0;

        //last child
        if(i == P-1){
                do{
                        rcnt = read(fd, buff, sizeof(buff));
                        if(rcnt == -1){
                                perror("read_in_child");
                                exit(1);
                        }
                        for(int j = 0; j < rcnt; ++j){
                                if(buff[j] == c) { ++cnt;}
                        }
                }while(rcnt != 0);

                printf("%d: I counted %d\n", i, cnt);
                return cnt;
        }

        //the other children
        int space;
        while(check < seg){
                space = sizeof(buff);
                if(space > (seg-check)) space = (seg-check);
                rcnt = read(fd, buff, space);
                if(rcnt == -1){
                        perror("read_in_child");
                        exit(1);
                }
                for(int j = 0; j < rcnt; ++j){
                        if(buff[j] == c) { ++cnt; }
                }
                check += rcnt;
        }
        printf("%d: I counted %d\n", i, cnt);
        return cnt;
}
```

Το πρόγραμμα λειτουργεί όπως αναμένεται:

```
oslab026@os-node1:~$ ./temp book2 book u
^C
6 children processes are still searching
5: I counted 23
4: I counted 19
^C
4 children processes are still searching
^C
4 children processes are still searching
3: I counted 19
^C
3 children processes are still searching
2: I counted 20
^C
2 children processes are still searching
1: I counted 19
^C
1 children processes are still searching
0: I counted 20
```

Μέρος 4ο: Εφαρμογή παράλληλης καταμέτρησης χαρακτήρων

Στα πλαίσια του 4ου μέρους της εργασίας μας, δημιουργήσαμε μια δομημένη εφαρμογή παράλληλης καταμέτρησης χαρακτήρων, η οποία μπορεί να δέχεται εντολές από τον χρήστη κατά την εκτέλεση της. Συγκεκριμένα, ο χρήστης μπορεί να προσθέσει ή να αφαιρέσει x διεργασίες αναζήτησης. Μπορεί, επιπλέον, να ενημερωθεί για την πρόοδο της αναζήτησης ή τον αριθμό και την “ταυτότητα” των διεργασιών που συμμετέχουν σε αυτή. Τα παραπάνω να επιτυγχάνει με τις αντίστοιχες τέσσερις εντολές: (i) “add x workers”, (ii) “remove x workers”, (iii) “show progress”, (iv) “show process info”.

Αποτελείται από τα παρακάτω ανεξάρτητα, αλλά συνεργαζόμενα, προγράμματα, τα οποία είναι αποθηκευμένα σε διαφορετικά αρχεία:

1. **Front-end**
   Είναι υπεύθυνος για την επικοινωνία με τον χρήστη. Ανάλογα με το είδος του αιτήματος του χρήστη, το ικανοποιεί ή το προωθεί στον dispatcher.
2. **Dispatcher**
   Αναλαμβάνει την κατανομή της εργασίας και τη διαχείριση των εργατών. Επικοινωνεί μαζί τους και προωθεί τα αποτελέσματα της αναζήτησης τους στο front-end.
3. **Workers**
   Ελέγχει επαναληπτικά ένα τμήμα του αρχείου ανάγνωσης για εμφανίσεις του χαρακτήρα αναζήτησης και ενημερώνει τον dispatcher για τα αποτελέσματα της μέτρησης.

Ο αριθμός των workers στο πρόγραμμα μας πρέπει να παραμένει μεγαλύτερος ή ίσος του 0 και, ταυτοχρονα, να μην υπερβαίνει τον αριθμό P, ο οποίος έχει οριστεί ως 16 με #define.

Ανάλυση λειτουργίας προγράμματος

Front-end

Αρχικά, ο front-end ενημερώνει τον χρήστη για τα αποδεκτά όρια του αριθμού εργατών.

Ο front-end επικοινωνεί με τον dispatcher, τον οποίον γεννά με fork() και σχηματίζει με execv(), μέσω τριών pipes pfd1, pfd2, pfd3. Το pfd1 έχει κατεύθυνση από τον front-end προς τον dispatcher και σκοπός του είναι η ενημέρωση του δεύτερου για τις εντολές που λαμβάνονται από τον χρήστη. Τα pfd2 και pfd3 κατευθύνονται από τον dispatcher προς τον front-end. Μέσω του pfd2 μεταφέρονται οι πληροφορίες για την πρόοδο της αναζήτησης (προς ικανοποίηση “show

progress”), ενώ το pfd3 για την αποστολή του τελικού αποτελέσματος μετά το πέρας των αναζητήσεων του συνόλου των εργατών.

Ο front-end βρίσκεται σε ένα while loop. Σε κάθε επανάληψη του, ελέγχει αν υπάρχει κάποιο μήνυμα στο unblocked standard input. Αν η ανάγνωση είναι επιτυχής, η διαχείριση είναι ανάλογη του μηνύματος. Αναλυτικότερα:
i) Αν αναγνωριστεί η εντολή “show process info”, καλείται η συνάρτηση show_pstree(), η οποία αξιοποιεί την κλήση συστήματος system() για να εκτελέσει την εντολή pstree στο shell. Η εντολή pstree λαμβάνει ως παράμετρο το PID του dipatcher. Εμφανίζει, λοιπόν, στον χρήστη το δέντρο διεργασιών με ρίζα τον dispatcher.
ii) Αν αναγνωριστεί η εντολή “show progress”, ο front-end την μεταφέρει στον dispatcher μέσω του σωλήνα pfd1. Περιμένει να διαβάσει διαδοχικά το ποσοστό του αρχείου που έχει ελεγχθεί και τον αριθμό των χαρακτήρων αναζήτησης που έχουν βρεθεί ως τώρα από τον σωλήνα pfd2. Όταν συλλέξει αυτες τις πληροφορίες, τις εκτυπώνει στο standard output.
Κάθε άλλο μήνυμα προωθείται αυτούσιο στον dispatcher μέσω του pfd2.

Το παραπάνω while loop έχει flag ως συνθήκη. Το συγκεκριμένο flag είναι μια int μεταβλητή που αρχικοποιείται σε 1 (true). Το flag μηδενίζεται, αποτιμάται δηλαδή ως false, μόνο όταν σταλεί από τον dispatcher σήμα SIGUSR1. Αυτό συμβαίνει όταν ο dispatcher καταλάβει ότι έχει ολοκληρωθεί η αναζήτηση. Η μεταβλητή flag δηλώνεται ως global για να μπορεί να μηδενιστεί εντός της συνάρτησης handler() που δημιουργούμε για το σήμα SIGUSR1 εντός του front-end.

Μετά τον τερματισμό του loop, ο front-end διαβάζει από τον σωλήνα pfd3 τον συνολικό αριθμό εμφανίσεων του χαρακτήρα. Εκτυπώνει το τελικό αποτέλεσμα της αναζήτησης στο standard output.

Dispatcher

Ο dispatcher είναι υπεύθυνος για την δημιουργία και το τερματισμό των workers, καθώς και για τον καταμερισμό εργασίας σε αυτούς. Λαμβάνει τα αποτελέσματα που έχουν παράξει οι workers και τα προωθεί τον front-end. ‘Οταν ο dispatcher ξεκινάει την εκτέλεση του πραγματοποιεί μια σειρά από λειτουργίες προτού βρεθεί σε θέση να δεχθεί εντολές μέσω του front-end.

Αρχικά ανοίγει το αρχείο προς αναζήτηση και υπολογίζει το μέγεθός του, αποθηκεύοντας το στο πεδίο st_size ενός struct τύπου stat μέσω της fstat(). Αυτό είναι αναγκαίο για τον αποδοτικό κατακερματισμό εργασίας στους workers. Πιο συγκεκριμένα, το αρχείο μοιράζεται σε N μέρη (size/100 μέσω #define N 100) και αργότερα, ο κάθε worker αναλαμβάνει ένα segment j του αρχείου, ψάχνοντας από το σημείο j*size/N μέχρι το (j+1)*size/N. ΄Υστερα, ο dispatcher δημιουργεί 2 pipes για κάθε worker, δηλαδή 2*P pipes, ένα για την ανάγνωση αποτελεσμάτων και ένα για την αποστολή του νέου segment προς αναζήτηση. Τέλος, ο dispatcher ορίζει τους

πίνακες jobs[N], staff[P] και worker_job[P]. Ο πίνακας jobs αρχικοποιείται με 0 για όλες τις θέσεις και ενημερώνεται όταν ένα segment έχει διαβαστεί ή είναι υπο αναζήτηση {0→unsearched, 1→searched, -1→under_search}. Ο πίνακας staff αρχικοποιείται με -1 και ενημερώνεται όταν προστίθεται ο n-οστός worker ως staff[n-1]=pid_of_n_worker. Τον πίνακα αυτόν τον χρησιμοποιούμε όταν θέλουμε να τερματίσουμε workers, μετά από εντολή remove ή μετά την ολοκλήρωση της αναζήτησης, στέλνοντας SIGKILL σε αυτούς μέσω των pid που βρίσκουμε στον staff. Ο πίνακας worker_job ενημερώνεται με βάση ποιο segment του αρχείου ανέλαβε ο n-οστός worker και μας βοηθάει για να ενημερώνουμε σωστά τον πίνακα jobs.

Ο dispatcher είναι πλέον έτοιμος να δεχθεί και να εκτελέσει εντολές από τον front-end. Μπαίνει λοιπόν σε ένα ατέρμονο loop το οποίο κάνει unblocked read από τον front-end για τυχόν εντολές. Αν η ανάγνωση είναι επιτυχής προσπαθεί να την ταιριάξει σε μία από τις τρεις περιπτώσεις των “add x workers”, “remove x workers” και “show progress”. Μετά, κάνει unblocked read από όσους workers υπάρχουν. Αν κάποια ανάγνωση είναι επιτυχής, δηλαδή ο worker έχει αναζητήσει όλο το segment που του είχε ανατεθεί, τότε συναθροίζει το αποτέλεσμα που διάβασε στον μετρητή χαρακτήρων προς αναζήτηση και αναθέτει στον worker νέο unsearched segment αν υπάρχει, ψάχνοντας το jobs για μηδενική τιμή (unsearched). Τέλος, ελέγχει αν έχουν διαβαστεί όλα τα segment ώστε να κάνει exit, διαφορετικά ξανακάνει unblocked read από τον front-end και συνεχίζει με αυτόν τον τρόπο.

Για το “show progress” ο dispatcher διαιρεί το πλήθος των χαρακτήρων που έχει διαβαστεί με το μέγεθος του αρχείου και το στέλνει μέσω pipe στον front-end. Έπειτα, του στέλνει το πλήθος των χαρακτήρων προς αναζήτηση που έχουν βρεθεί. Για το “add x workers”, ο dispatcher δημιουργεί επαναληπτικά x παιδιά με fork() και με execv τα κάνει workers, περνώντας τους και το πρώτο segment που πρέπει να διαβάσουν και ενημερώνοντας κατάλληλα τους πίνακες. Για το “remove x workers”, ο dispatcher στέλνει SIGKILL διαδοχικά x φορές στους workers που βρίσκονται τέλος τους πίνακα staff ανεξαρτήτως αν αυτοί έχουν αναλάβει segment ή όχι. Αν έχουν αναλάβει, τότε ο πίνακας jobs στην θέση worker_job του αντίστοιχου worker γίνεται 0 από -1, και διαβάζεται και το pipe που αντιστοιχεί στον worker για να αδειάσει και να μην προκαλέσει προβλήματα στον επομενο worker που θα πάρει την θέση του.

Μόλις διαβαστούν όλα τα segments, ο dispatcher στέλνει στον front-end το progress (δηλαδή 100%) και τον τελικό αριθμό χαρακτήρων που βρέθηκαν για να καλύψει την περίπτωση που ο front-end ζήτησε “show progress” αλλά ο dispatcher τερμάτισε προτού το ικανοποιήσει. Δεν στέλνουμε, δηλαδή, τον τελικό αριθμό χαρακτήρων μονο στον σωλήνα που αφορά το τελικό αποτέλεσμα, αλλά και στον σωλήνα που αφορά τα αποτελέσματα της εντολής “show progress”.

Worker

Η αναζήτηση γίνεται από τους workers με κλήση της συνάρτησης search(), η οποία λειτουργεί όμοια με τη συνάρτηση child() του 3ου μέρους.

Κάθε worker ξεκινά τη ζωή του αναζητώντας χαρακτήρες στο τμήμα του αρχείου που του ανατίθεται αρχικά από τον dispatcher. Στέλνει το αποτέλεσμα της αναζήτησης στον dispatcher, μέσω του σωλήνα με κατεύθυνση από αυτόν προς τον dispatcher. Εισέρχεται, έπειτα, σε έναν ατέρμονο βρόχο. Σε κάθε επανάληψη, περιμένει ώσπου να διαβάσει από τον σωλήνα με κατεύθυνση από τον dispatcher προς αυτόν την επόμενη ανάθεση. Προχωρά στην αναζήτηση του νέου τμήματος και αποστολή του αποτελέσματος στον dispatcher.

Σχόλια

Τα file descriptors μπορούν να κληρονομηθούν ακόμα και όταν γίνεται execv. Για να χρησιμοποιηθεί το file descriptor “pfd” ενός pipe μεταξύ του παιδιού και του γονέα πρέπει να γίνει dup2(pfd,1), ή κάποιος άλλος θετικός ακέραιος αριθμός, πριν γίνει το execv, στο κομμάτι κώδικα που εκτελείται από το παιδί μετά το fork(). Ύστερα, το παιδί μπορεί να κάνει write(1,...) για να γράψει στο pipe. Αυτό είναι μια εναλλακτική προσέγγιση από το να περάσουμε τους file descriptors σαν arguments κατά το execv. Στην υλοποίηση μας, αξιοποιήθηκαν και οι δύο τρόποι. Συγκεκριμένα, η dup2 χρησιμοποιήθηκε για να ανακτηθούν οι file descriptors των σωλήνων επικοινωνίας του dispatcher με τους workers.

Για τον έλεγχο του προγράμματος παρακάτω χρησιμοποιείται το αρχείο book2 που περιέχει 120 u.

```
oslab026@os-node1:~$ cat book2
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
uuuuuuuuuuuu
```

Έλεγχος προγράμματος

Όπως φαίνεται στο παρακάτω στιγμιότυπο, το πρόγραμμα έχει την επιθυμητή λειτουργία.

```
oslab026@os-node1:~$ ./a1.4-front-end book2 u
Add workers to begin.
The number of workers will remain between 0 and 16 at all times.
Given a command that suggests otherwise, the closest value to that will be applied.
add 10 workers
The character u appears in file book2 120 times.
```

Για να μπορέσουμε να ελέγξουμε την ανταπόκριση του προγράμματος στις εντολές “remove workers”, “show progress” και “show process info”, χρειάζεται να εισάγουμε τεχνητές καθυστερήσεις. Επιλέγουμε να καθυστερήσουμε κατά 2 second την έναρξη της αναζήτησης κάθε κομματιού.

```
int search(int fd, char c, long size, int i){
        sleep(2);
```

Επιπλέον, θα φροντίσουμε έτσι ώστε να εκτυπώνονται ενδείξεις για την γέννηση και τον θανάτο κάθε worker. Θα γίνει ορατή, έτσι, η εσωτερική κατάσταση του προγράμματος.

Ένδειξη γέννησης worker:

```c
int main(int argc, char *argv[]){
        int fd = atoi(argv[1]);
        char c = argv[2][0];
        int pcnt = atoi(argv[3]);
        int pass = atoi(argv[4]);
        long size = atol(argv[5]);
        int i = atoi(argv[6]);
        int k;
        printf("worker %ld was born\n", (long)getpid());
        int res = search(fd, c, size, i);
        //printf("got the jpb\n");
        write(pcnt, &res, sizeof(res));
        while(1){
                read(pass, &k, sizeof(k));
                //printf("%d->%d\n", i, k);
                int res = search(fd, c, size, k);
                write(pcnt, &res, sizeof(res));
        }
}
```

Ένδειξη ότι ο dispatcher τερμάτισε την εκτέλεση κάποιου worker:

```c
   //remove workers
   else if((sscanf(buff, "remove %d %7s", &x, w) == 2) && (strcmp(w, "workers") == 0)){
           buff[0] = '\0';
           int sink;
           //printf("master set me free\n");
           for(int i = 0; (i < x) && (ind > 0); i++){
                   if(worker_job[ind-1] >= 0) jobs[worker_job[ind-1]] = 0;
                   kill(staff[ind-1], SIGKILL);
                   waitpid(staff[ind-1], &status, 0);
                   printf("dispatcher killed process %d\n", ind-1);
                   read(pcnt[ind-1][0], &sink, sizeof(sink));//empty the pipe
                   ind--;
           }
   }
```

Το πρόγραμμα λειτουργεί όπως αναμένεται.

```
oslab026@os-node1:~$ ./a1.4-front-end book2 u
Add workers to begin.
The number of workers will remain between 0 and 16 at all times.
Given a command that suggests otherwise, the closest value to that will be applied.
add 4 workers
worker 1847472 was born
worker 1847473 was born
worker 1847475 was born
worker 1847474 was born
show process info


a1.4-dispatcher(1847466)qwqa1.4-worker(1847472)
                        tqa1.4-worker(1847473)
                        tqa1.4-worker(1847474)
                        mqa1.4-worker(1847475)


remove 2 workers
dispatcher killed process 3
dispatcher killed process 2
show process info


a1.4-dispatcher(1847466)qwqa1.4-worker(1847472)
                        mqa1.4-worker(1847473)


show progress
46.153847% work done, characters found so far = 56
add 2 workers
worker 1847498 was born
worker 1847499 was born
show process info


a1.4-dispatcher(1847466)qwqa1.4-worker(1847472)
                        tqa1.4-worker(1847473)
                        tqa1.4-worker(1847498)
                        mqa1.4-worker(1847499)


show progress
75.384613% work done, characters found so far = 91
The character u appears in file book2 120 times.
```

Το μόνο πρόβλημα που αντιμετωπίζουμε είναι η εμφάνιση της pstree. Η κωδικοποίηση των γραμμών στο δέντρο διεργασιών εμφανίζει συμβολοσειρές, πράγμα που δεν συνέβαινε σε προηγούμενες δοκιμές. Δοκιμάσαμε να περάσουμε τις παραμέτρους -U, -A στην pstree, αντί για την -G, αλλά το πρόβλημα δεν λύθηκε. Πιθανότητα υπάρχει κάποια αναντιστοιχία στην κωδικοποίηση σε βαθύτερο επίπεδο του terminal.