|  |  |
| --- | --- |
|  | **ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ**  ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ  ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ |

ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΝΑΦΟΡΑ 2Η

Από τους:

|  |  |
| --- | --- |
| Ιωάννης Μιχαήλ Καζελίδης  Α.Μ.: 03117885 | Μάριος Κερασιώτης  Α.Μ.: 03117890 |

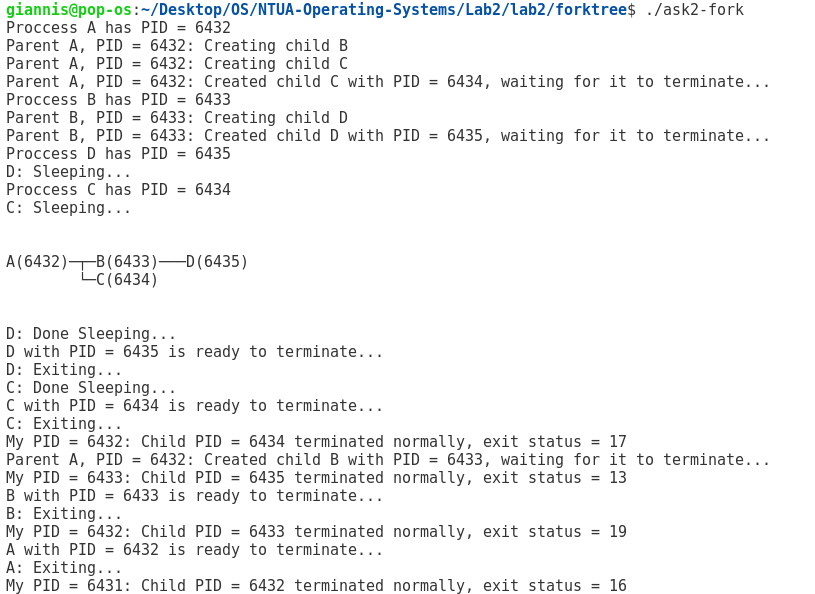
# ΑΣΚΗΣΕΙΣ

## Δημιουργία δεδομένου δέντρου διεργασιών

Για αυτό το ερώτημα χρησιμοποιούμε τον παρακάτω πηγαίο κώδικα. Ταυτόχρονα χρησιμοποιούμε τα αρχεία proc-common.{c, h} που μας είναι δοσμένα.

|  |
| --- |
| #include <assert.h>  #include <stdio.h>  #include <stdlib.h>  #include <sys/types.h>  #include <sys/wait.h>  #include <unistd.h>  #include "proc-common.h"  #define SLEEP\_PROC\_SEC 10  #define SLEEP\_TREE\_SEC 3  #define EXIT\_A 16  #define EXIT\_B 19  #define EXIT\_C 17  #define EXIT\_D 13  /\*   \* Create this process tree:   \* A-+-B---D   \*   `-C   \*/  void fork\_procs(void) {    /\*     \* initial process is A.     \*/    pid\_t pid\_b, pid\_c, pid\_d;    int status;    change\_pname("A");    printf("Proccess A has PID = %ld\n", (long)getpid());    // Code bellow is for child B    fprintf(stderr, "Parent A, PID = %ld: Creating child B\n", (long)getpid());    pid\_b = fork();    if (pid\_b < 0) {      /\* fork failed \*/      perror("B: fork");      exit(1);    }    if (pid\_b == 0) {      change\_pname("B");      printf("Proccess B has PID = %ld\n", (long)getpid());      fprintf(stderr, "Parent B, PID = %ld: Creating child D\n", (long)getpid());      pid\_d = fork();      if (pid\_d < 0) {        /\* fork failed \*/        perror("D: fork");        exit(1);      }      if (pid\_d == 0) {        change\_pname("D");        printf("Proccess D has PID = %ld\n", (long)getpid());        printf("D: Sleeping...\n");        sleep(SLEEP\_PROC\_SEC);        printf("D: Done Sleeping...\n");        printf("D with PID = %ld is ready to terminate...\nD: Exiting...\n",               (long)getpid());        exit(EXIT\_D);      }      printf(          "Parent B, PID = %ld: Created child D with PID = %ld, waiting for it "          "to terminate...\n",          (long)getpid(), (long)pid\_d);      pid\_d = wait(&status);      explain\_wait\_status(pid\_d, status);      printf("B with PID = %ld is ready to terminate...\nB: Exiting...\n",             (long)getpid());      exit(EXIT\_B);    }    // Code bellow is for child C    fprintf(stderr, "Parent A, PID = %ld: Creating child C\n", (long)getpid());    pid\_c = fork();    if (pid\_c < 0) {      /\* fork failed \*/      perror("C: fork");      exit(1);    }    if (pid\_c == 0) {      change\_pname("C");      printf("Proccess C has PID = %ld\n", (long)getpid());      printf("C: Sleeping...\n");      sleep(SLEEP\_PROC\_SEC);      printf("C: Done Sleeping...\n");      printf("C with PID = %ld is ready to terminate...\nC: Exiting...\n",             (long)getpid());      exit(EXIT\_C);    }    printf(        "Parent A, PID = %ld: Created child C with PID = %ld, waiting for it to "        "terminate...\n",        (long)getpid(), (long)pid\_c);    pid\_c = wait(&status);    explain\_wait\_status(pid\_c, status);    printf(        "Parent A, PID = %ld: Created child B with PID = %ld, waiting for it to "        "terminate...\n",        (long)getpid(), (long)pid\_b);    pid\_b = wait(&status);    explain\_wait\_status(pid\_b, status);    printf("A with PID = %ld is ready to terminate...\nA: Exiting...\n",           (long)getpid());    exit(EXIT\_A);  }  /\*   \* The initial process forks the root of the process tree,   \* waits for the process tree to be completely created,   \* then takes a photo of it using show\_pstree().   \*   \* How to wait for the process tree to be ready?   \* In ask2-{fork, tree}:   \*      wait for a few seconds, hope for the best.   \*/  int main(void) {    pid\_t pid;    int status;    /\* Fork root of process tree \*/    pid = fork();    if (pid < 0) {      perror("main: fork");      exit(1);    }    if (pid == 0) {      /\* Child \*/      fork\_procs();      exit(1);    }    /\*     \* Father     \*/    /\* for ask2-{fork, tree} \*/    sleep(SLEEP\_TREE\_SEC);    /\* Print the process tree root at pid \*/    show\_pstree(pid);    /\* Wait for the root of the process tree to terminate \*/    pid = wait(&status);    explain\_wait\_status(pid, status);    return 0;  } |

Και τρέχοντας το πρόγραμμα λαμβάνουμε έξοδο:

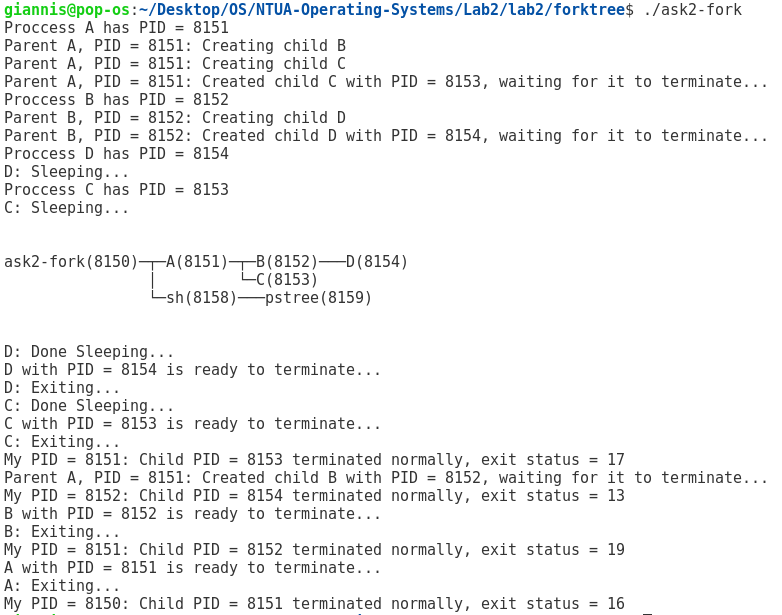


### Ερώτηση 1η

Η διεργασία Α είναι πατέρας των B, C. Εάν την τερματίσουμε πρόωρα δηλαδή την τερματίσουμε πριν οι διεργασίες B, C ολοκληρωθούν τότε οι διεργασίες αυτές θα γίνουν “zombie” διεργασίες και θα γίνουν παιδιά της init (PID = 1) που εκτελεί συνεχώς wait().

### Ερώτηση 2η

Το αποτέλεσμα της show\_pstree(getpid()) φαίνεται παρακάτω:



Από τα man pages μπορούμε να καταλάβουμε ότι η getpid() επιστρέφει το PID της καλούμενης διεργασίας που στη περίπτωση μας η διεργασία αυτή είναι η ask2-fork (PID=8150). Ως αποτέλεσμα η show\_pstree() τυπώνει ένα δέντρο που έχει ρίζα όχι τη διεργασία Α αλλά τη διεργασία μέσα στην οποία βρισκόμαστε τώρα, την ask2-fork. Αξιοσημείωτο είναι ότι μέσα στο δέντρο προστέθηκε η διεργασία sh (PID=8158) και η pstree (PID=8159), που είναι παιδί της. Αυτό γίνεται διότι η show\_pstree για να λειτουργήσει καλεί μέσω της διεργασίας του bash (sh (PID = 8158)) την pstree (PID = 8159). Για αυτόν τον λόγο επομένως η ask2-fork καλεί την διεργασία του bash, η οποία με την σειρά της καλεί την pstree, δημιουργώντας έτσι το παραπάνω σχήμα.

### Ερώτηση 3η

Ο εκάστοτε διαχειριστής ενός συστήματος πρέπει να εξασφαλίσει ότι κανένας χρήστης δεν θα μπορεί να δημιουργήσει αυθαίρετο αριθμό από διεργασίες, αφήνοντας έτσι ανοικτή την περίπτωση εκούσια (π.χ. κάποιος ιός) ή ακούσια (π.χ. ατελείωτη αναδρομή) να κάνει κατάχρηση των διαθέσιμων πόρων του συστήματος με αποτέλεσμα το σύστημα να γίνει ασταθές.

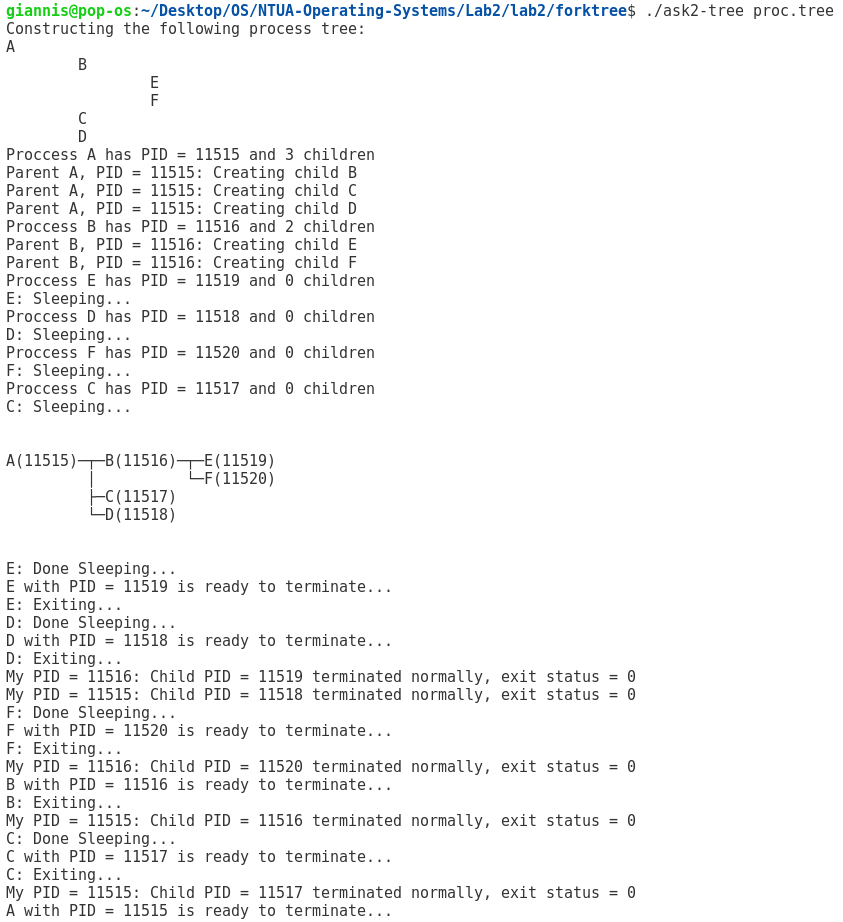
Ένα χαρακτηριστικό παράδειγμα εκούσιας κατάχρησης πόρων αποτελεί η επίθεση forkbomb η οποία δημιουργεί συνεχώς αντίγραφα του εαυτού της με σκοπό να κάνει ένα σύστημα να καταρρεύσει (crash-άρει) (DoS : Denial of Service attack).

## Δημιουργία αυθαίρετου δέντρου διεργασιών

Για αυτό το ερώτημα χρησιμοποιούμε τον παρακάτω πηγαίο κώδικα. Ταυτόχρονα χρησιμοποιούμε τα αρχεία proc-common.{c, h}, tree.{c, h} που μας είναι δοσμένα.

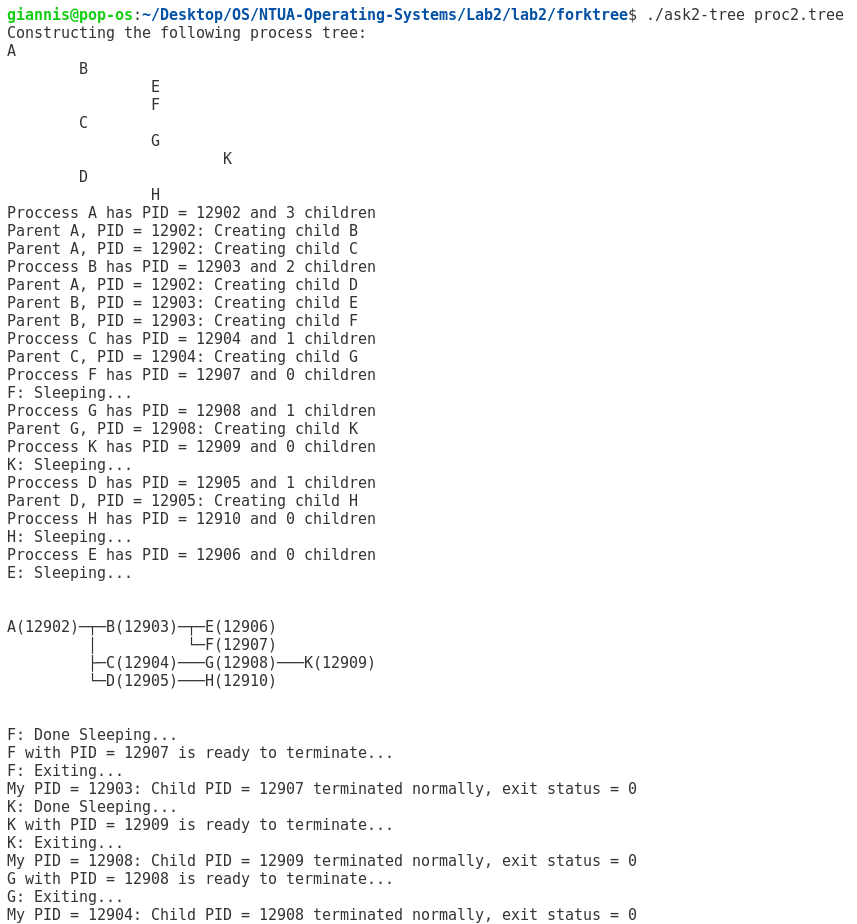
|  |
| --- |
| #include <unistd.h>  #include <stdio.h>  #include <stdlib.h>  #include <assert.h>  #include <sys/types.h>  #include <sys/wait.h>  #include "proc-common.h"  #include "tree.h"  #define SLEEP\_PROC\_SEC 10  #define SLEEP\_TREE\_SEC 3  void fork\_procs(struct tree\_node \*node)  {      pid\_t child;      int status;      int i;      change\_pname(node->name);      printf("Proccess %s has PID = %ld and %d children\n", node->name, (long)getpid(), node->nr\_children);      for (i = 0; i < node->nr\_children; ++i)      {          fprintf(stderr, "Parent %s, PID = %ld: Creating child %s\n", node->name, (long)getpid(), node->children[i].name);          child = fork();          if (child < 0)          {              /\* fork failed \*/              perror("Error at children");              exit(1);          }          if (child == 0)          {              fork\_procs(&node->children[i]);          }      }      for (i = 0; i < node->nr\_children; ++i)      {          child = wait(&status);          explain\_wait\_status(child, status);      }      if (node->nr\_children == 0)      {          printf("%s: Sleeping...\n", node->name);          sleep(SLEEP\_PROC\_SEC);          printf("%s: Done Sleeping...\n", node->name);      }      printf("%s with PID = %ld is ready to terminate...\n%s: Exiting...\n", node->name, (long)getpid(), node->name);      exit(0);  }  /\* The initial process forks the root of the process tree,   \* waits for the process tree to be completely created,   \* then takes a photo of it using show\_pstree().   \*`   \* How to wait for the process tree to be ready?   \* In ask2-{fork, tree}:   \*      wait for a few seconds, hope for the best.   \*/  int main(int argc, char \*argv[])  {      pid\_t pid;      int status;      struct tree\_node \*root;      if (argc != 2)      {          fprintf(stderr, "Usage: %s <input\_tree\_file>\n\n", argv[0]);          exit(1);      }      root = get\_tree\_from\_file(argv[1]);      printf("Constructing the following process tree:\n");      print\_tree(root);      /\* Fork root of process tree \*/      pid = fork();      if (pid < 0)      {          perror("main: fork");          exit(1);      }      if (pid == 0)      {          /\* Child \*/          fork\_procs(root);          exit(1);      }      /\* for ask2-{fork, tree} \*/      sleep(SLEEP\_TREE\_SEC);      /\* Print the process tree root at pid \*/      show\_pstree(pid);      /\* Wait for the root of the process tree to terminate \*/      pid = wait(&status);      explain\_wait\_status(pid, status);      return 0;  } |

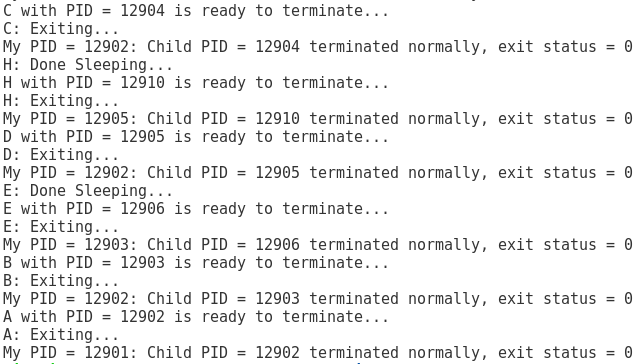
Τρέχοντας το πρόγραμμα με είσοδο το αρχείο proc.tree που μας δίνεται λαμβάνουμε έξοδο:



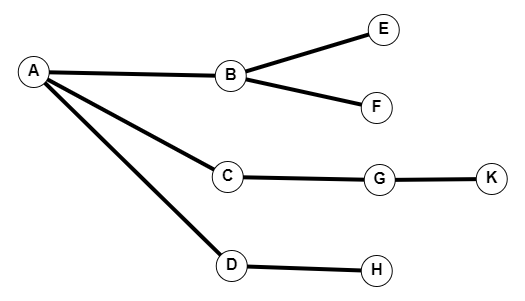


Τρέχοντας το πρόγραμμα με είσοδο το αρχείο proc2.tree που δημιουργήσαμε λαμβάνουμε έξοδο:





Όπου proc2.tree:



### Ερώτηση 1η

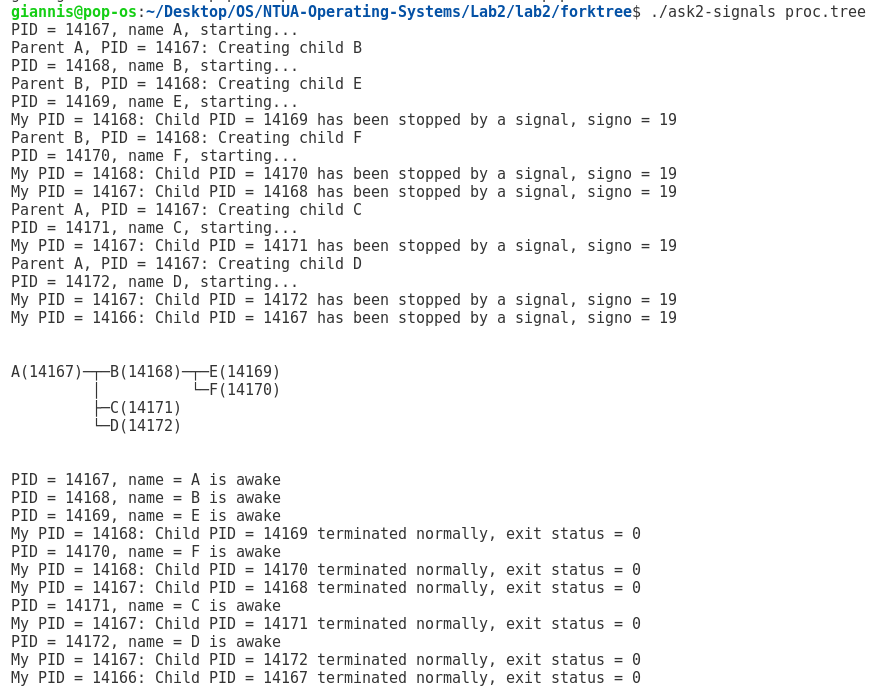
Το πρόγραμμα αρχικά δημιουργεί τους κόμβους αρχίζοντας από τους γονείς και εκ των υστέρων δημιουργεί και τα παιδιά-φύλλα, επομένως η δημιουργία γίνεται με top-down λογική (η δημιουργία των διεργασιών ακολουθεί δηλαδή τον αλγόριθμο BFS καθώς διατρέχουμε το δέντρο διεργασιών εισόδου κατά επίπεδα). Αντίθετα, τερματισμός γίνεται με bottom-up λογική, καθώς θέλουμε να τερματιστούν πρώτα τα φύλλα και μετά οι εσωτερικοί κόμβοι, διότι θέλουμε να πεθάνουν ομαλά τα παιδιά και να μην μείνουν ορφανά.

## Αποστολή και χειρισμός σημάτων

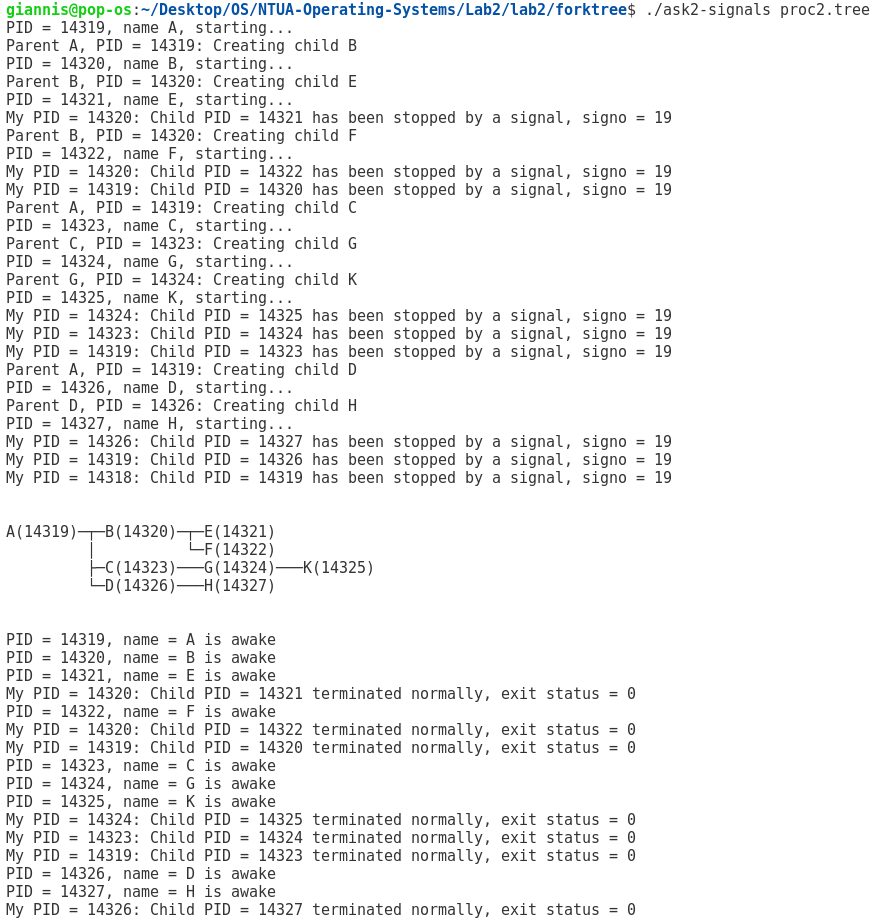
Για αυτό το ερώτημα χρησιμοποιούμε τον παρακάτω πηγαίο κώδικα. Ταυτόχρονα χρησιμοποιούμε τα αρχεία proc-common.{c, h}, tree.{c, h} που μας είναι δοσμένα.

|  |
| --- |
| #include <unistd.h>  #include <stdio.h>  #include <stdlib.h>  #include <assert.h>  #include <signal.h>  #include <sys/types.h>  #include <sys/wait.h>  #include "tree.h"  #include "proc-common.h"  void fork\_procs(struct tree\_node \*root)  {      /\*       \* Start       \*/      int i;      int status[root->nr\_children];      pid\_t child[root->nr\_children];      printf("PID = %ld, name %s, starting...\n",             (long)getpid(), root->name);      change\_pname(root->name);      if (root->nr\_children == 0)      {                   // is leaf          raise(SIGSTOP); // stop what you are doing until parent requests it          printf("PID = %ld, name = %s is awake\n",                 (long)getpid(), root->name);      }      else      {          for (i = 0; i < root->nr\_children; ++i)          {              fprintf(stderr, "Parent %s, PID = %ld: Creating child %s\n", root->name, (long)getpid(), root->children[i].name);              child[i] = fork();              if (child[i] < 0)              {                  /\* fork failed \*/                  perror("Error at children");                  exit(1);              }              if (child[i] == 0)              {                  fork\_procs(&root->children[i]);                  exit(1);              }              wait\_for\_ready\_children(1);          }          /\*          \* Suspend Self          \*/          raise(SIGSTOP);          printf("PID = %ld, name = %s is awake\n",                 (long)getpid(), root->name);          for (i = 0; i < root->nr\_children; ++i)          {              kill(child[i], SIGCONT);              waitpid(child[i], &status[i], 0);              explain\_wait\_status(child[i], status[i]);          }      }      /\*       \* Exit       \*/      exit(0);  }  /\*   \* The initial process forks the root of the process tree,   \* waits for the process tree to be completely created,   \* then takes a photo of it using show\_pstree().   \*   \* In ask2-signals:   \*      use wait\_for\_ready\_children() to wait until   \*      the first process raises SIGSTOP.   \*/  int main(int argc, char \*argv[])  {      pid\_t pid;      int status;      struct tree\_node \*root;      if (argc < 2)      {          fprintf(stderr, "Usage: %s <tree\_file>\n", argv[0]);          exit(1);      }      /\* Read tree into memory \*/      root = get\_tree\_from\_file(argv[1]);      /\* Fork root of process tree \*/      pid = fork();      if (pid < 0)      {          perror("main: fork");          exit(1);      }      if (pid == 0)      {          /\* Child \*/          fork\_procs(root);          exit(1);      }      /\*       \* Father       \*/      /\* for ask2-signals \*/      wait\_for\_ready\_children(1);      /\* Print the process tree root at pid \*/      show\_pstree(pid);      /\* for ask2-signals \*/      kill(pid, SIGCONT);      /\* Wait for the root of the process tree to terminate \*/      wait(&status);      explain\_wait\_status(pid, status);      return 0;  } |

Τρέχοντας το πρόγραμμα με είσοδο το αρχείο proc.tree που μας δίνεται λαμβάνουμε έξοδο:



Τρέχοντας το πρόγραμμα με είσοδο το αρχείο proc2.tree που δημιουργήσαμε λαμβάνουμε έξοδο:





### Ερώτηση 1η

Στις προηγούμενες ασκήσεις αδρανοποιούσαμε τα παιδιά ώστε να εξασφαλίσουμε χρόνο πριν τερματιστούν ώστε να προλάβουν να αποτυπωθούν από την pstree που καλείται από την show\_pstree(). Με την χρήση σημάτων αποφεύγουμε να ρυθμίζουμε αυθαίρετα τον χρόνο όπως κάναμε στην 1.2 με την sleep(), πετυχαίνοντας καλύτερο συγχρονισμό. Συγκεκριμένα μόλις “κοιμηθούν” με την raise(SIGSTOP) και αφού εμφανίσει το δέντρο στέλνει μήνυμα στα παιδιά με το kill(child\_pid, SIGCONT) να ξυπνήσουν και να συνεχίσουν έως ότου τερματιστούν.

### Ερώτηση 2η

|  |
| --- |
| void wait\_for\_ready\_children(int cnt)  {      int i;      pid\_t p;      int status;      for (i = 0; i < cnt; i++) {          /\* Wait for any child, also get status for stopped children \*/          p = waitpid(-1, &status, WUNTRACED);          explain\_wait\_status(p, status);          if (!WIFSTOPPED(status)) {              fprintf(stderr, "Parent: Child with PID %ld has died unexpectedly!\n",                  (long)p);              exit(1);          }      }  } |

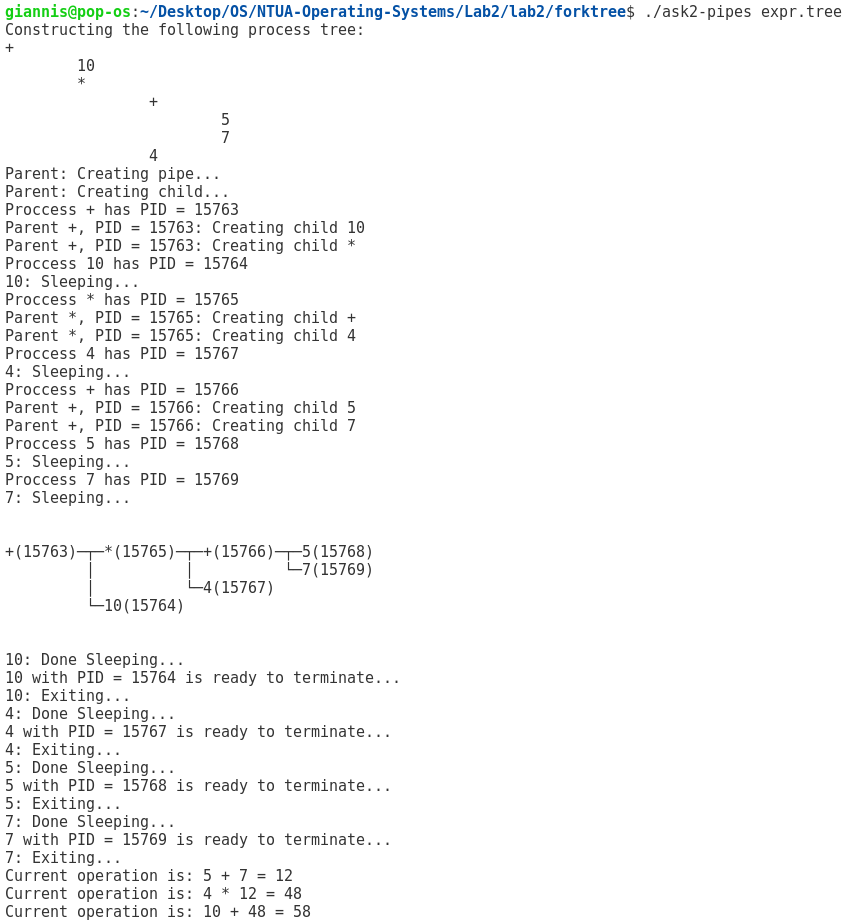
Όπως βλέπουμε από τον παραπάνω κώδικα η wait\_for\_ready\_children() είναι μια συνάρτηση που αναστέλλει την λειτουργία της μέχρι τα παιδιά της να αδρανοποιηθούν. Ελέγχει ουσιαστικά τη σημαία WIFSTOPPED, που δείχνει αν το παιδί έχει σταματήσει και σε αντίθετη περίπτωση εμφανίζει μήνυμα για τον «ξαφνικό» θάνατο του παιδιού και κάνει exit(). Η χρήση της εξασφαλίζει πως όλα τα παιδιά είναι ζωντανά και έχουν σταματήσει, άρα το δέντρο διεργασιών θα εμφανιστεί σωστά και επίσης ότι θα έχουμε πλήρη γνώση αν κάποιο παιδί πέθανε πριν την ώρα του. Αν δε τη χρησιμοποιήσουμε, τότε ο πατέρας αφού ξυπνήσει θα περιμένει το θάνατο του παιδιού του, που έχει ήδη προηγηθεί, επομένως θα παραμένει αδρανής η διεργασία.

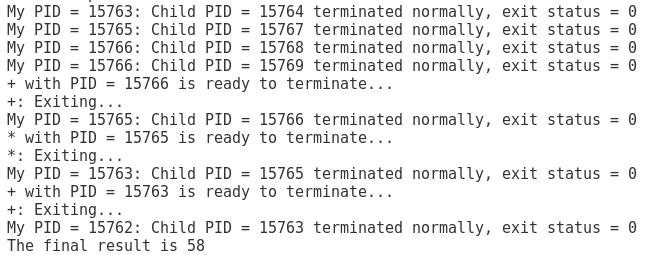
## Παράλληλος υπολογισμός αριθμητικής έκφρασης

Για αυτό το ερώτημα χρησιμοποιούμε τον παρακάτω πηγαίο κώδικα. Ταυτόχρονα χρησιμοποιούμε τα αρχεία proc-common.{c, h}, tree.{c, h} που μας είναι δοσμένα.

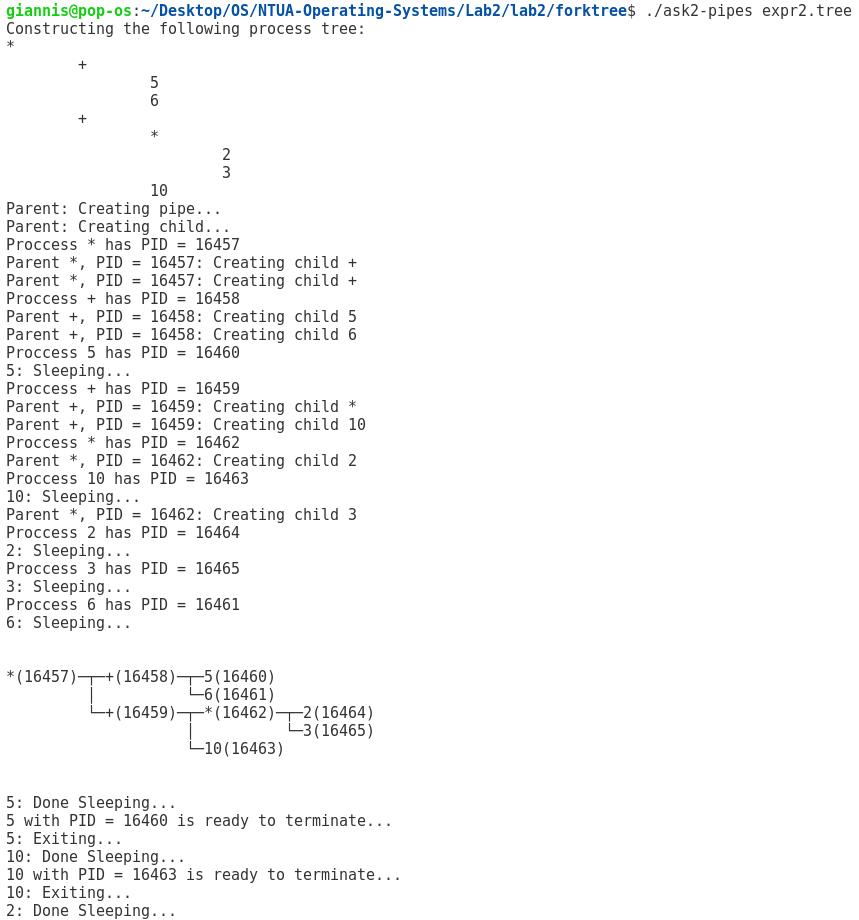
|  |
| --- |
| #include <unistd.h>  #include <stdio.h>  #include <stdlib.h>  #include <sys/types.h>  #include <sys/wait.h>  #include <signal.h>  #include <string.h>  #include "proc-common.h"  #include "tree.h"  #define SLEEP\_PROC\_SEC 10  #define SLEEP\_TREE\_SEC 3  void fork\_procs(struct tree\_node \*node, int pfd[])  {      pid\_t child;      int status;      int i;      change\_pname(node->name);      printf("Proccess %s has PID = %ld\n", node->name, (long)getpid());      if (node->nr\_children == 0)      {          printf("%s: Sleeping...\n", node->name);          sleep(SLEEP\_PROC\_SEC);          printf("%s: Done Sleeping...\n", node->name);          if (close(pfd[0]) < 0)          {              perror("closing pipe");              exit(1);          }          int val = atoi(node->name);          if (write(pfd[1], &val, sizeof(val)) != sizeof(int))          {              perror("writing in pipe");              exit(1);          }      }      else      {          int operators[2] = {0, 0};          int newpipe[2];          if (pipe(newpipe) < 0)          {              perror("pipe");              exit(1);          }          for (i = 0; i < 2; ++i)          {              fprintf(stderr, "Parent %s, PID = %ld: Creating child %s\n", node->name, (long)getpid(), node->children[i].name);              child = fork();              if (child < 0)              {                  /\* fork failed \*/                  perror("Error at children");                  exit(1);              }              if (child == 0)              {                  fork\_procs(&node->children[i], newpipe);              }          }          if (close(newpipe[1]) < 0)          {              perror("closing pipe");              exit(1);          }          for (i = 0; i < 2; i++)          {              if (read(newpipe[0], &operators[i], sizeof(operators[i])) != sizeof(operators[i]))              {                  perror("Reading pipe");                  exit(1);              }          }          int res;          if (!strcmp(node->name, "+"))          {              res = operators[0] + operators[1];              printf("Current operation is: %d + %d = %d\n", operators[0], operators[1], res);          }          else          {              res = operators[0] \* operators[1];              printf("Current operation is: %d \* %d = %d\n", operators[0], operators[1], res);          }          if (close(pfd[0]) < 0)          {              perror("closing pipe");              exit(1);          }          if (write(pfd[1], &res, sizeof(res)) != sizeof(res))          {              perror("writing in pipe");              exit(1);          }          for (i = 0; i < 2; ++i)          {              child = wait(&status);              explain\_wait\_status(child, status);          }      }      printf("%s with PID = %ld is ready to terminate...\n%s: Exiting...\n", node->name, (long)getpid(), node->name);      exit(0);  }  int main(int argc, char \*argv[])  {      pid\_t pid;      int status;      int pfd[2];      struct tree\_node \*root;      if (argc != 2)      {          fprintf(stderr, "Usage: %s <input\_tree\_file>\n\n", argv[0]);          exit(1);      }      root = get\_tree\_from\_file(argv[1]);      printf("Constructing the following process tree:\n");      print\_tree(root);      printf("Parent: Creating pipe...\n");      if (pipe(pfd) < 0)      {          perror("pipe");          exit(1);      }      printf("Parent: Creating child...\n");      pid = fork();      if (pid < 0)      {          /\* fork failed \*/          perror("main: fork");          exit(1);      }      if (pid == 0)      {          fork\_procs(root, pfd);          exit(1);      }      /\*       \* In parent process.       \*/      sleep(SLEEP\_TREE\_SEC);      show\_pstree(pid);      int res;      if (read(pfd[0], &res, sizeof(res)) != sizeof(res))      {          perror("read from pipe");          exit(1);      }      /\* Print the process tree root at pid \*/      pid = wait(&status);      explain\_wait\_status(pid, status);      printf("The final result is %d\n", res);      return 0;  } |

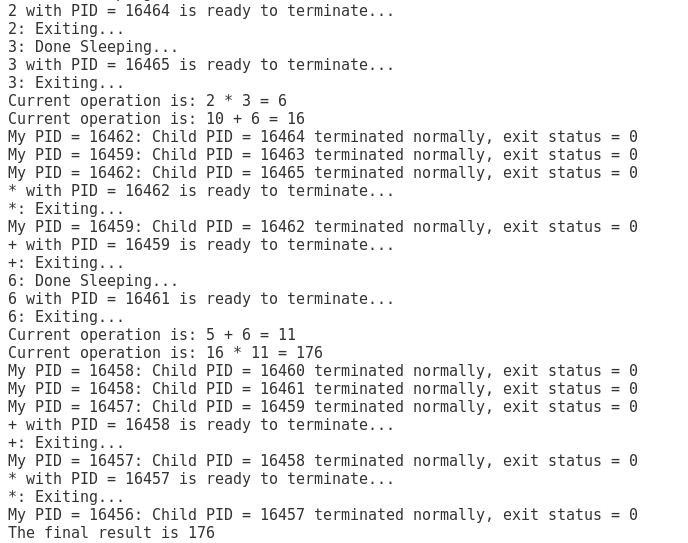
Τρέχοντας το πρόγραμμα με είσοδο το αρχείο expr.tree που μας δίνεται λαμβάνουμε έξοδο:



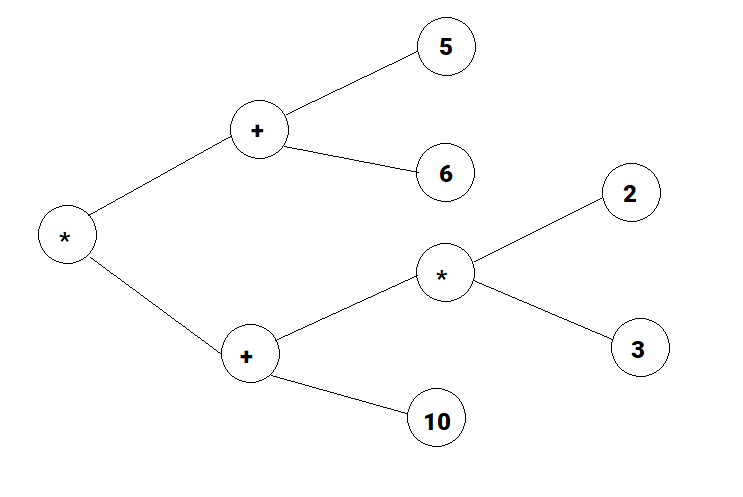


Τρέχοντας το πρόγραμμα με είσοδο το αρχείο expr2.tree που δημιουργήσαμε λαμβάνουμε έξοδο:





Όπου είναι expr2.tree το:



### Ερώτηση 1η

Στην άσκηση γίνεται η χρήση μία σωλήνωσης ανά διεργασία. Γενικά, αυτό είναι δυνατό γιατί τα writes μέσα στο pipe γίνονται ατομικά ,το οποίο σημαίνει ότι αν δύο η περισσότερα παιδιά προσπαθούν να γράψουν μέσα στο ίδιο pipe δεν θα υπάρχει περίπτωση να υπάρξει πρόβλημα με μπερδεμένα δεδομένα (εναλλάξ δεδομένα από τα διαφορετικά παιδιά). Συγκεκριμένα, είναι δυνατό διότι το δέντρο διασχίζεται κατά βάθος και φτάνοντας στα φύλλα με το μεγαλύτερο βάθος αρχίζουν να γίνονται οι πράξεις της πρόσθεσης και του πολλαπλασιασμού, που είναι αντιμεταθετικές πράξεις. Στην περίπτωση που γινόντουσαν πράξεις όπως αφαίρεση και διαίρεση, που δεν ισχύει η αντιμεταθετική ιδιότητα, θα χρειαζόντουσαν τουλάχιστον δύο pipes.

### Ερώτηση 2η

Σε ένα τέτοιο σύστημα μπορούν να τρέξουν ταυτόχρονα περισσότερες διεργασίες, αν αυτές δεν είναι αλληλεξαρτώμενες. Συγκεκριμένα σε αυτό το πρόγραμμα θα εκτελούνται παράλληλα οι διεργασίες παιδιά κάθε γονιού. Αυτό σημαίνει ότι μπορούμε στον ίδιο χρόνο να κάνουμε περισσότερες πράξεις και να λαμβάνουμε γρηγορότερα το τελικό αποτέλεσμα. Αν δεν έχουμε τέτοιο σύστημα αναγκαζόμαστε να ακολουθούμε το ρυθμό του επεξεργαστή και τη σειρά εντολών του κώδικα.