

ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

Δραστηριότητα 1

ΜΠΑΣΑΓΙΑΝΝΗ ΓΕΩΡΓΙΑ 1084016

ΠΑΝΟΥΡΓΙΑΣ ΑΝΤΩΝΙΟΣ 1083996

ΣΤΕΡΓΙΟΠΟΥΛΟΣ ΓΕΩΡΓΙΟΣ 1083861

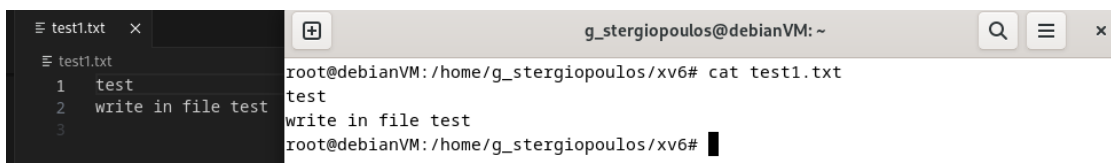
Άσκηση 1

Οι κλήσεις συστήματος ορίζονται στο αρχείο user.h (xv6\include\user.h) του πηγαίου κώδικα.

```
include > C user.h
1 struct stat;
2 struct pstat;
3
4 // system calls
5 int fork(void);
6 int exit(void) __attribute__((noreturn));
7 int wait(void);
8 int pipe(int*);
9 int write(int, void*, int);
10 int read(int, void*, int);
11 int close(int);
12 int kill(int);
13 int exec(char*, char**);
14 int open(char*, int);
15 int mknod(char*, short, short);
16 int unlink(char*);
17 int fstat(int fd, struct stat*);
18 int link(char*, char*);
19 int mkdir(char*);
20 int chdir(char*);
21 int dup(int);
22 int getpid(void);
23 char* sbrk(int);
24 int sleep(int);
25 int uptime(void);
26 int getpinfo(struct pstat*);
```

Από τα προγράμματα χρήστη στον φάκελο user διαλέγουμε το cat που βρίσκεται στο αρχείο με path: user\cat.c. Το συγκεκριμένο πρόγραμμα έχει τις εξής προσφερόμενες λειτουργίες (ανάλογα με την χρήση της):

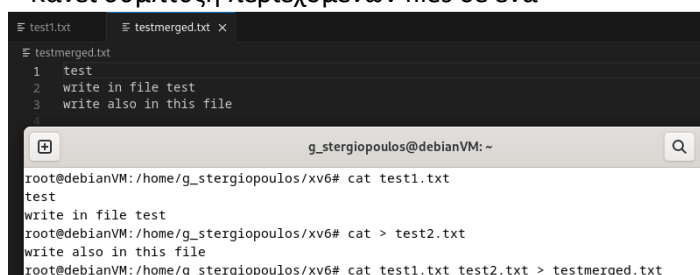
- Διαβάζει περιεχόμενα ενός file και τα τυπώνει στο terminal



```
g_stergiopoulos@debianVM: ~
root@debianVM: /home/g_stergiopoulos/xv6# cat test1.txt
test
write in file test
root@debianVM: /home/g_stergiopoulos/xv6#
```

- Αντιγράφει περιεχόμενα ενός file σε ένα άλλο

- Κάνει σύμπτυξη περιεχομένων files σε ένα



```
g_stergiopoulos@debianVM: ~
root@debianVM: /home/g_stergiopoulos/xv6# cat test1.txt
test
write in file test
root@debianVM: /home/g_stergiopoulos/xv6# cat > test2.txt
write also in this file
root@debianVM: /home/g_stergiopoulos/xv6# cat test1.txt test2.txt > testmerged.txt
```

- Δημιουργεί file και προσφέρει την δυνατότητα εγγραφής από το standard input στο file

```

test1.txt x
test1.txt
1 test
2 write in file test
3
root@debianVM: /home/g_stergiopoulos/xv6# cat
test
test
echo
echo
^C
root@debianVM: /home/g_stergiopoulos/xv6# cat > test1.txt
test
write in file test

```

- Διαβάζει το standard input και τυπώνει ξανά στο standard output

```

g_stergiopoulos@debianVM: ~
root@debianVM: /home/g_stergiopoulos/xv6# cat
test
test
echo
echo

```

Συγκεκριμένα μελετώντας τον πηγαίο κώδικα μπορούμε να κατανοήσουμε τον τρόπο λειτουργίας του καλύτερα και ποιες **system calls** αξιοποιεί.

```

user > C cat.c
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4
5  char buf[512];
6
7  void
8  cat(int fd)
9  {
10     int n;
11
12     while((n = read(fd, buf, sizeof(buf))) > 0)
13         write(1, buf, n);
14     if(n < 0){
15         printf(1, "cat: read error\n");
16         exit();
17     }
18 }
19

```

```

20 int
21 main(int argc, char *argv[])
22 {
23     int fd, i;
24
25     if(argc <= 1){
26         cat(0);
27         exit();
28     }
29
30     for(i = 1; i < argc; i++){
31         if((fd = open(argv[i], 0)) < 0){
32             printf(1, "cat: cannot open %s\n", argv[i]);
33             exit();
34         }
35         cat(fd);
36         close(fd);
37     }
38     exit();
39 }

```

Στην main: Περνάμε ως arguments, έναν πίνακα string (argv) που δίνει ο χρήστης από το standard input και το argc που δείχνει πόσα strings περιέχονται στο argv.

Χρησιμοποιώντας την **open()** προσπαθεί να αποκτήσει πρόσβαση στο αρχείο που δίνει ο χρήστης και επιστρέφει έναν ακέραιο file descriptor ο οποίος θα αξιοποιηθεί από τις read και write αργότερα για να διευκρινίσει το που θα γίνει το input και το output αντίστοιχα.

Με την **read()** παίρνει την πληροφορία από το συγκεκριμένο fd και την βάζει στο buffer (το input δεν μπορεί να ξεπεράσει τα 512 bytes). Στην συνέχεια επιστρέφει το μέγεθος του περιεχομένου του buffer στην μεταβλητή n.

Από την άλλη η **write()** μπορεί να περάσει την πληροφορία που περάσαμε στο buffer, στο output που ορίζει ο fd = 1.

Με την `close()` απελευθερώνουμε το συγκεκριμένο fd το καθιστώντας το ανοιχτό για χρήση από το υπόλοιπο σύστημα.

Στο τέλος του προγράμματος γίνεται χρήση της `exit()` για να τερματίσει η διεργασία.

Άσκηση 2

Στην άσκηση αυτή θα δημιουργήσουμε ένα νέα user program, το dog.c. Η πολύ χρήσιμη δουλειά που εκτελεί είναι το να διαβάζει ένα input string από τον χρήστη και να το επιστρέφει πίσω σε αλφαβητική σειρά. Το πρόγραμμα τερματίζει όταν δεχθεί είσοδο 0 από τον χρήστη! (Προφανώς προσθέσαμε το πρόγραμμα και στην λίστα USER_PROGS του Makefile)

```
user > C dog.c
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4
5  char buf[512];
6
7  void
8  dog(int fd)
9  {
10     int n;
11
12     while((n = read(fd, buf, sizeof(buf))) > 0){
13         if(buf[0]=='\0'){
14             printf(1, "\n #####GOOD BYE##### \n");
15             exit();
16         }
17
18         char temp;
19         int i,j;
20         int k = strlen(buf);
21         for(i=0; i<k-1; i++){
22             for(j=i+1; j<k; j++){
23                 if(buf[i]>buf[j]){
24                     temp = buf[i];
25                     buf[i] = buf[j];
26                     buf[j] = temp;
27                 }
28             }
29         }
30     }
31 }
```

```
33     printf(1, "Sorted String: %s\nTry another string: ", buf);
34     for(i=0; i<k; i++)
35     {
36         buf[i] = '\0';
37     }
38 }
39
40 if(n < 0){
41     printf(1, "dog: read error\n");
42     exit();
43 }
44 }
45
46 int
47 main(int argc, char *argv[])
48 {
49     int fd, i, j;
50     printf(1, "Give a string to be sorted (0 for exit):");
51
52     if(argc <= 1){
53         dog(0);
54         exit();
55     }
56
57     for(i = 1; i < argc; i++){
58         if((fd = open(argv[i], 0)) < 0){
59             printf(1, "dog: cannot open %s\n", argv[i]);
60             exit();
61         }
62         dog(fd);
63         close(fd);
64     }
65     exit();
66 }
```

Παράδειγμα χρήσης:

```
cpu0: starting xv6
cpu1: starting
cpu0: starting
init: starting sh
$ dog
Give a string to be sorted (0 for exit):fajfhahfue
Sorted String:
aaeffffhjhju
Try another string: eiaf
Sorted String:
aefi
Try another string: 0
#####GOOD BYE#####
```

Άσκηση 3

Εντοπίζουμε την system call, uptime, στα αρχεία syscall.c και sysproc.c. Στο δεύτερο είναι και εκεί που υλοποιείται:

```
114 // Return how many clock tick interrupts have occurred
115 // since start.
116 int
117 sys_uptime(void)
118 {
119     uint xticks;
120
121     acquire(&tickslock);
122     xticks = ticks;
123     release(&tickslock);
124     return xticks;
125 }
```

Η uptime τυπώνει: την ακριβή ώρα, πόσο χρόνο τρέχει το σύστημα, πόσοι χρήστες χρησιμοποιούν το σύστημα και τον μέσο αριθμό διεργασιών που τρέχουν τα τελευταία 1, 5 και 15 λεπτά.

Παράδειγμα χρήσης:

```
root@debianVM: /home/g_stergiouopoulos/xv6# uptime
14:30:36 up 2 days, 21:04,  1 user,  load average: 2.14, 0.70, 0.47
```

Ο τρόπος υλοποίησής της είναι:

Με το acquire βάζει ένα spinlock σε μία διεύθυνση για την αποφυγή δυσλειτουργίας κάποιου thread και σταματάει τα interrupts.

Παίρνει ύστερα το πόσα ticks έχουν περάσει από την αρχή και τέλος κάνοντας το release απελευθερώνει τον καταχωρητή για να συνεχίσει το πρόγραμμα.

Γενικά για τέτοιες λειτουργίες χρησιμοποιούμε το ζεύγος acquire-release στην αρχή και το τέλος αντίστοιχα για να επιτυγχάνεται fencing και να γίνεται lock. Έτσι μετριοούνται τα ticks στην περίπτωση μας.

Άσκηση 4

Όταν καλούμε ένα system call η διαδικασία είναι η εξής:

Από το user.h παίρνουμε τα γνωρίσματα syscalls και εκτελείται αυτόματα η δημιουργία του usys.S με το system call number από το syscall.h. Με ένα usertrap μεταφερόμαστε από user mode σε kernel mode. Τη στιγμή αυτή ελέγχει αν πρόκειται για TIMER, SYSCALL, DEVICE ή EXCEPTION. Όταν βρει ότι πρόκειται για syscall, αναζητά τον κωδικό της συνάρτησης που έχει αποθηκεύσει τον καταχωρητή eax για να βρει ποια θα καλέσει. Τέλος βάσει αυτού του κωδικού εκτελείται η εκάστοτε system call κανονικά. Όταν τελειώσει η εκτέλεσή της με usertrapret και την εντολή return επιστρέφει και πάλι σε user mode.

Άσκηση 5

Προσθέτουμε μία νέα κλήση συστήματος που θα επιστρέφει τον αγαπημένο μας αριθμό int getfanum() , τον 26113. Τα βήματα που χρειάζεται να ακολουθήσουμε είναι τα εξής:

- Προσθέτουμε την συνάρτηση στα header user.h και syscall.h

```
include > C user.h
1 struct stat;
2 struct pstat;
3
4 // system calls
5 int fork(void);
6 int exit(void) __attribute__((noreturn));
7 int wait(void);
8 int pipe(int*);
9 int write(int, void*, int);
10 int read(int, void*, int);
11 int close(int);
12 int kill(int);
13 int exec(char*, char**);
14 int open(char*, int);
15 int mknod(char*, short, short);
16 int unlink(char*);
17 int fstat(int fd, struct stat*);
18 int link(char*, char*);
19 int mkdir(char*);
20 int chdir(char*);
21 int dup(int);
22 int getpid(void);
23 char* sbrk(int);
24 int sleep(int);
25 int uptime(void);
26 int getpinfo(struct pstat*);
27 int getfavnum(void);
28 void halt(void);
29 int getcount(int);
```

```
include > C syscall.h
1 // System call numbers
2 #define SYS_fork 1
3 #define SYS_exit 2
4 #define SYS_wait 3
5 #define SYS_pipe 4
6 #define SYS_read 5
7 #define SYS_kill 6
8 #define SYS_exec 7
9 #define SYS_fstat 8
10 #define SYS_chdir 9
11 #define SYS_dup 10
12 #define SYS_getpid 11
13 #define SYS_sbrk 12
14 #define SYS_sleep 13
15 #define SYS_uptime 14
16 #define SYS_open 15
17 #define SYS_write 16
18 #define SYS_mknod 17
19 #define SYS_unlink 18
20 #define SYS_link 19
21 #define SYS_mkdir 20
22 #define SYS_close 21
23 #define SYS_getpinfo 24
24 #define SYS_getfavnum 23
```

- Την προσθέτουμε και στις λίστες στην syscall.c καθώς και στο αντίστοιχο σημείο στην usys.S

```
126 extern int sys_chdir(void);
127 extern int sys_close(void);
128 extern int sys_dup(void);
129 extern int sys_exec(void);
130 extern int sys_exit(void);
131 extern int sys_fork(void);
132 extern int sys_fstat(void);
133 extern int sys_getpid(void);
134 extern int sys_kill(void);
135 extern int sys_link(void);
136 extern int sys_mkdir(void);
137 extern int sys_mknod(void);
138 extern int sys_open(void);
139 extern int sys_pipe(void);
140 extern int sys_read(void);
141 extern int sys_sbrk(void);
142 extern int sys_sleep(void);
143 extern int sys_unlink(void);
144 extern int sys_wait(void);
145 extern int sys_write(void);
146 extern int sys_uptime(void);
147 extern int sys_getpinfo(void);
148 extern int sys_getfavnum(void);
149 extern void sys_halt(void);
150 extern int sys_getcount(int syscall);
```

```
152 static int (*syscalls[])(void) = {
153 [SYS_fork] sys_fork,
154 [SYS_exit] sys_exit,
155 [SYS_wait] sys_wait,
156 [SYS_pipe] sys_pipe,
157 [SYS_read] sys_read,
158 [SYS_kill] sys_kill,
159 [SYS_exec] sys_exec,
160 [SYS_fstat] sys_fstat,
161 [SYS_chdir] sys_chdir,
162 [SYS_dup] sys_dup,
163 [SYS_getpid] sys_getpid,
164 [SYS_sbrk] sys_sbrk,
165 [SYS_sleep] sys_sleep,
166 [SYS_uptime] sys_uptime,
167 [SYS_open] sys_open,
168 [SYS_write] sys_write,
169 [SYS_mknod] sys_mknod,
170 [SYS_unlink] sys_unlink,
171 [SYS_link] sys_link,
172 [SYS_mkdir] sys_mkdir,
173 [SYS_close] sys_close,
174 [SYS_getpinfo] sys_getpinfo,
175 [SYS_getfavnum] sys_getfavnum,
176 [SYS_halt] sys_halt,
177 [SYS_getcount] sys_getcount,
178 };
```

```
ulib > usys.S
4 #define SYSCALL(name) \
5 .globl name; \
6 name: \
7 movl $SYS_ ## name, %eax; \
8 int $T_SYSCALL; \
9 ret
10
11 SYSCALL(fork)
12 SYSCALL(exit)
13 SYSCALL(wait)
14 SYSCALL(pipe)
15 SYSCALL(read)
16 SYSCALL(write)
17 SYSCALL(close)
18 SYSCALL(kill)
19 SYSCALL(exec)
20 SYSCALL(open)
21 SYSCALL(mknod)
22 SYSCALL(unlink)
23 SYSCALL(fstat)
24 SYSCALL(link)
25 SYSCALL(mkdir)
26 SYSCALL(chdir)
27 SYSCALL(dup)
28 SYSCALL(getpid)
29 SYSCALL(sbrk)
30 SYSCALL(sleep)
31 SYSCALL(uptime)
32 SYSCALL(getpinfo)
33 SYSCALL(getfavnum)
34 SYSCALL(halt)
35 SYSCALL(getcount)
```

- Τέλος την υλοποιούμε στο sysproc.c

```
127 int
128 sys_getfavnum(void)
129 {
130
131 return 26113;
132 }
```

Για να την εκτελέσουμε πρέπει να την χρησιμοποιήσουμε μέσα σε ένα user prog

Παράδειγμα χρήσης καλώντας την μέσα στην dog:

```
$ dog
0 KALUTEROS ARITHMOS: 26113,
```

Άσκηση 6

Θέλουμε να υλοποιήσουμε μία νέα system call που θα κάνει shutdown το μηχάνημά μας, συγκεκριμένα το qemu. Την ορίζουμε ως void halt(void) με την ίδια διαδικασία όπως και πριν.

Η υλοποίησή της γίνεται και πάλι στο sysproc.c

```
133
134 void
135 sys_halt(void)
136 {
137     outw(0x604, 0x2000);
138 }
139
```

Με την εντολή outw που περιέχεται στο header x86.h αλλάζουμε την τιμή στους συγκεκριμένους καταχωρητές που έχει ορίσει το qemu ως τους υπεύθυνους για την ένδειξη στο σύστημα ότι πρέπει να κάνει shutdown.

Για να την χρησιμοποιεί ο χρήστης γράφουμε και νέο user prog, το shutdown.c στο οποίο την καλούμε.

```
user > C shutdown.c
1 #include "types.h"
2 #include "stat.h"
3 #include "user.h"
4
5 int main()
6 {
7     halt(); ←
8     exit();
9     return 0;
10 }
```

```
cpu0: starting xv6
cpu1: starting
cpu0: starting
init: starting sh
$ shutdown
root@debianVM: /home/g_stergiopoulos/xv6#
```

Άσκηση 7

Στην άσκηση αυτή προσθέτουμε μία νέα system call, την int getcount(int syscall) που θα δίνει την δυνατότητα στον χρήστη να ελέγξει πόσες φορές έχει κληθεί μια system call.

Αρχικά ακολουθούμε την γνωστή διαδικασία δημιουργίας νέας κλήσης συστήματος την int getcount(int syscall). Η λογική υλοποίησής της είναι ότι κάθε φορά που καλείται η syscall συνάρτηση από το αρχείο syscall.c θα καλεί μία άλλη νέα συνάρτηση την counter_array(int code) που ορίζουμε στο sysproc.c με όρισμα τον κωδικό της system call η οποία θα αυξάνει ένα array από counters, το counter_arr[] στην εκάστοτε θέση ανάλογα με τον κωδικό που

δίνει. Τέλος ορίζουμε το πρόγραμμα χρήστη counter.c που διαβάζει το input του χρήστη και θα καλεί την νέα system call που θα μας επιστρέφει την απάντηση που ζητάμε.

```
180 void
181 syscall(void)
182 {
183     int num;
184
185     num = proc->tf->eax;
186     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
187         proc->tf->eax = syscalls[num]();
188         //////////
189         counter_array(num);
190         //cprintf("the numver is:%d \n",num);
191         //counter_array[num]++;
192     } else {
193         cprintf("%d %s: unknown sys call %d\n",
194             proc->pid, proc->name, num);
195         proc->tf->eax = -1;
196     }
197 }
```

Οι αλλαγές που κάναμε στην syscall

```
140 int counter_array(int code){
141     counter_arr[code]++;
142     return 0;
143 }
```

Η συνάρτηση που καλεί η syscall

```
145 int
146 sys_getcount(void)
147 {
148     int n;
149     argint(0, &n);
150     return counter_arr[n];
151 }
```

Η νέα system call

```
user > C counter.c
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4
5  int counter(int code);
6
7  int main(int argc, char *argv[])
8  {
9      int code,i;
10     for(i=1;i<argc;i++){
11         code = atoi(argv[i]);
12         counter(code);
13     }
14     return 0;
15 }
16
17
18
19 int counter(int code){
20
21     printf(1,"The system call with code %d, has been called %d times!\n", code,getcount(code));
22     return 0;
23 }
```

Το νέα user prog

Τους κωδικούς της κάθε system call μπορεί να τους βρει ο χρήστης στο αρχείο syscall.h:

```
include > C syscall.h
1  // System call numbers
2  #define SYS_fork 1
3  #define SYS_exit 2
4  #define SYS_wait 3
5  #define SYS_pipe 4
6  #define SYS_read 5
7  #define SYS_kill 6
8  #define SYS_exec 7
9  #define SYS_fstat 8
10 #define SYS_chdir 9
11 #define SYS_dup 10
12 #define SYS_getpid 11
13 #define SYS_sbrk 12
14 #define SYS_sleep 13
15 #define SYS_uptime 14
16 #define SYS_open 15
17 #define SYS_write 16
18 #define SYS_mknod 17
19 #define SYS_unlink 18
20 #define SYS_link 19
21 #define SYS_mkdir 20
22 #define SYS_close 21
23 #define SYS_getpinfo 24
24 #define SYS_getfavnum 23
25 #define SYS_halt 25
26 #define SYS_getcount 26
```

Παράδειγμα χρήσης:

```
cpu0: starting xv6
cpu1: starting
cpu0: starting
init: starting sh
$ counter 16 ←

The system call with code 16, has been called 20 times!
pid 3 counter: trap 14 err 4 on cpu 0 eip 0xffffffff add
$ counter 16

The system call with code 16, has been called 79 times!
pid 4 counter: trap 14 err 4 on cpu 1 eip 0xffffffff add
$ counter 5

The system call with code 5, has been called 32 times!
pid 5 counter: trap 14 err 4 on cpu 1 eip 0xffffffff add
```

Άσκηση 8

Τώρα προσθέτουμε μία νέα system call με πρότυπο `int killrandom(void)`, η οποία θα τερματίζει μία τυχαία διεργασία που εκτελείται.

Ακολουθώντας πάλι την γνωστή διαδικασία που περιγράφεται στις προηγούμενες ασκήσεις δημιουργούμε μία νέα system call την `sys_killrandom`. Ο κώδικας εκτελείται στο `sysproc.c` καλώντας την συνάρτηση `getranpid()` για να βρει μία τυχαία διεργασία που τρέχει καθώς επίσης την `getpidlist()` που βρίσκει τη λίστα των pid.

```
int
sys_killrandom(void)
{
    int ran_pid = getranpid();

    getpidlist();
    return 0;
}
```

Εδώ δίνεται μία εκδοχή κατά την οποία βρίσκουμε τα PID όλων των διεργασιών που τρέχουν στο σύστημα (κατάσταση `RUNNABLE`). Ενεργοποιούμε τις διακοπές και με τον `phtable` να είναι μέσα στα όρια του πλήθους των διεργασιών ελέγχεται μέχρι να βρει μία που είναι `RUNNABLE`. Η διαδικασία επιλογής τυχαίας διεργασίας γίνεται με την κλήση της `randomrange`.

```
int
getranpid(void){
    struct proc *p;
    for(;;){
        sti();
        acquire(&phtable.lock);
        for(p=phtable.proc;p<&phtable.proc[NPROC];p++)
        {
            if(p->state != RUNNABLE)
                continue;
            p->pid;
        }
        release(&phtable.lock);
    }
    int ran_pid;
    ran_pid = randomrange(0, NPROC);

    return ran_pid;
}
```


Ζητήθηκε να προστεθεί μία γεννήτρια τυχαίων αριθμών στο kernel και για το σκοπό αυτό χρησιμοποιούμε τις ακόλουθες συναρτήσεις random και randomrange. Με αυτόν τον μηχανισμό έχουμε κάθε φορά τυχαίο αριθμό διεργασίας μέσω όλων αυτών των πράξεων. Οι συναρτήσεις για την γεννήτρια τυχαίων αριθμών βρέθηκαν στο διαδίκτυο.

```
uint
random(void)
{
    static unsigned int z1 = 12345, z2 = 12345, z3 = 12345, z4 = 12345;
    unsigned int b;
    b = ((z1 << 6) ^ z1) >> 13;
    z1 = ((z1 & 4294967294U) << 18) ^ b;
    b = ((z2 << 2) ^ z2) >> 27;
    z2 = ((z2 & 4294967288U) << 2) ^ b;
    b = ((z3 << 13) ^ z3) >> 21;
    z3 = ((z3 & 4294967280U) << 7) ^ b;
    b = ((z4 << 3) ^ z4) >> 12;
    z4 = ((z4 & 4294967168U) << 13) ^ b;

    return (z1 ^ z2 ^ z3 ^ z4) / 2;
}

int
randomrange(int lo, int hi)
{
    if (hi < lo) {
        int tmp = lo;
        lo = hi;
        hi = tmp;
    }
    int range = hi - lo + 1;
    return random() % (range) + lo;
}
```